

Themis SPARC 10MP Software Manual

Version 2.0— March 20, 1996



Themis Computer—American and Pacific Rim
3185 Latentview Court
Fremont, CA 94538
Phone (510) 252-0870
Fax (510) 490-5529
Web Site: www.themis.com

Themis Computer—Rest of World
1, Rue Des Essarts
Z.A. De Mayencin
38610 Gieres, France
Phone 33 76 59 60 61
Fax 33 76 63 00 30

©Copyright 1995 Themis Computer Inc.

All rights reserved. No part of this work may be reproduced in any form or by any means without prior written permission by Themis Computer.

Restricted Rights Legend: Use, duplication or disclosure by the US government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the *Rights in Technical Data and Computer Software* clause of DFARS 252.227-7013 (October 1988) and FAR 52.227-19 (June 1987).

The information in this document has been carefully checked and is believed to be accurate. However, Themis Computer assumes no responsibility for inaccuracies. Themis computer retains the right to make changes to this document at any time without prior notice. Themis Computer does not assume any liability arising from the application or use of this document or the products(s) described herein.

UNIX[™] is a registered trademark of AT&T

OPENLOOK[™] is a registered trademark of AT&T and Sun Microsystems

SOLARIS[™] is a registered trademark of Sun Microsystems

SPARC[™], SPARC 10MP[™] are registered trademarks of SPARC International

The following are trademarks of their respective companies and organizations. Themis Computer disclaims any responsibility for specifying which marks are owned by which companies and organizations:

Advanced Micro Devices (AMD), ANSI, Centronics, IEEE, Intel, ISO, Ethernet, Fujitsu, JEDEC, Level One Communications, Mbus, NewBridge, NICE, SBus, SCSI, Signetics, SGS Thompson Microelectronics, Sun Microsystems, VMEbus, Zilog

Table of Contents

1: How to Use This Manual	1
1.1: Intended Audience	1
1.2: Organization	2
1.3: In Case Of Difficulties	3
2: Installing the SPARC IOMP Software	5
2.1: Installing THEMISvme	5
2.2: Detailed Installation	6
2.3: Removing THEMISvme	7
2.4: Installing THEMISvme On Diskless or Dataless Clients	8
2.5: Installing the Sample Drivers	8
2.6: Sample Installation of THEMISvme	9
2.7: Sample Installation of the Sample Drivers	11
3: Using the VME Interface	13
3.1: Read/Write Test	14
3.2: mmap Test	16
3.3: DVMA (slave mode access) test	18
3.4: Interrupts Test	23
3.5: DMA Test	26
4: Writing Programs for the VMEbus	31
4.1: Solaris 2.x Device Hierarchy	32
4.2: Features of the VMEbus	33
4.3: Configuring the Software Interface of Themis SPARC IOMP	34
4.4: Accessing the VMEbus from Solaris	34
4.4.1: Using Read/Write	35
4.4.2: Using mmap	37
4.4.3: DMA on the VMEbus	39
4.4.4: Performing a Slave Mode Access to Another VME Board	42
4.4.5: Advanced Features of the VMEbus Interface	44
5: Writing VME Device Drivers	45
5.1: Solaris 2.x Device Hierarchy	46
5.2: Features of the VMEbus	48

5.3: Configuration Files for VME Device Drivers	49
5.4: Probing devices.....	50
5.5: Registering Interrupts	52
5.6: Allocating DVMA Space.....	54
5.7: Mapping VMEbus Space.....	56
5.8: Driving Devices Without Writing Device Drivers	57
6: SPARC 10MP Programmers Guide.....	61
6.1: Introduction.....	61
6.2: Overview of VME Interface Under Solaris 1.1.1B/2.X	62
6.2.1: VMEbus Nexus Driver	63
6.2.2: Utility Drivers.....	64
6.2.3: Sample Drivers	65
7: Advanced Configuration of 10MP.....	67
7.1: Disabling VMEbus Interrupts.....	68
7.2: Isolating Boards from VMEbus SYSRESET	70
7.3: Handling Data Mismatch Over the VMEbus.....	71
7.4: Handling VMEbus Lock-Up Problems	72
7.5: Configuring VMEbus Release Mode.....	73
7.6: Installing Solaris 1.x Patches on Diskless Clients	75
7.7: Removing NEXUS Driver	91
7.8: Determining Speed of MBus Modules	93
7.9: Memory Error Message	94
7.10: VME Bus Interrupt Problem.....	116
7.11: 10MP OBP PROM	118
7.12: New 10MP OBP PROM Reset	119
7.13: Using TTYC OBP PROM	120
8: Sample Device Drivers	121

The Themis Computer SPARC 10MP is a SPARC-based single-board VMEbus computer. The software interface for the SPARC 10MP is transparently implemented under Solaris 1.x and Solaris 2.x and is compatible with Sun's SPARCstation 10 workstation. Themis Computer has also developed custom software that enables software programmers to effectively use the powerful features of the SPARC 10MP.

1.1 Intended Audience

The custom software containing programs, documentation and packaging, is targeted for different kinds of software users:

- System Administrators who install the software and perform the necessary software configuration.
- Users who perform day-to-day operations on SPARC 10MP systems.
- Application programmers who write user-level programs that utilize the VMEbus interface provided under Solaris.
- System programmers / device driver writers who develop kernel-level device drivers for VMEbus devices.

Some of these functions overlap one another. The basic concepts required for many of these functions are common. This manual is structured around the basic concepts of using a VMEbus system.

1.2 Organization

This document is organized in chapters that focus on a specific functional use of the VMEbus software interface.

- **Chapter 1, How to Use This Manual:** contains a brief overview of this document and its organization.
- **Chapter 2, Installing the SPARC 10MP Software:** contains instructions on installing the many parts of the software interface on a Solaris system.
- **Chapter 3, Using the VME Interface:** is intended for users of SPARC 10MP systems. It provides instructions on using the various user level programs provided by Themis Computer. These programs could be used to test the various features of the VME interface. The programs themselves are provided along with the source code and can be modified by the users of the system. The programs can be found in the 'example' subdirectory.
- **Chapter 4, Writing Programs for the VMEbus:** is intended for programmers who are not familiar with the Solaris architecture and the concepts of the VMEbus. The document discusses the features of Solaris drivers and VMEbus devices as they pertain to the Themis SPARC 10MP product and provides tips on writing programs to utilize the different features of the VMEbus.
- **Chapter 5, Writing VME Device Drivers:** is intended for systems programmers writing device drivers for VMEbus devices. The guide provides basic information on VMEbus specific features that influence the design of drivers for VMEbus devices. The guide also contains tips on programming VMEbus devices without writing custom device drivers.
- **Chapter 6, Programmers Manual:** contains extensive information useful to experienced programmers who would use the SPARC 10MP under Solaris Operating Environment.
- **Chapter 7, Advanced Configuration of SPARC 10MP:** the SPARC 10MP product provides powerful advanced features like generating a SYSRESET on the VMEbus, configuring the interrupt mechanism etc. Themis Computer periodically issues Technical Notes that detail these interfaces and ways to access them. These Technical Notes have been put together in this document.

- **Chapter 8, Sample Device Drivers:** these drivers have been written to illustrate features particular to VMEbus device drivers. These drivers are provided along with the source code. The manual pages for the drivers contain sample programs that use the drivers.
- **Appendix A, Manual Pages:** online manual pages have been added for all drivers provided by Themis Computer. Manual pages are available for the standard nexus and DMA drivers as well as the sample drivers.

1.3 In Case Of Difficulties

If the 10MP does not behave as described or if you encounter difficulties installing or configuring the 10MP either standalone or as part of your VMEbus system environment, please call Themis Computer technical support at 510-252-0870, fax your questions to 510-490-5529, or e-mail them to support@themis.com. You can also contact us at our web site: www.themis.com.



Installing the SPARC 10MP Software

2

The software interface for SPARC 10MP is distributed as a software package for Solaris 2.x systems. The software package is named **THEMISvme** and can be installed and removed like other standard Solaris software packages, by using the **pkgadd** and **pkgrm** commands.

2.1 Installing THEMISvme

The **THEMISvme** package is distributed on standard media and can be installed directly from the media. To install the package, place the installation media in the appropriate slot and execute this command:

```
# pkgadd -d <media name>
```

where **<media name>** is the name of the media on your system.

The **pkgadd** command will copy the contents of the package to the appropriate directories and perform the necessary installation. After the command completes, the system has to be rebooted for the new software to be operational.

The **THEMISvme** package can be installed safely on a system that already has a previous version of SPARC 10MP drivers installed. The previous versions of the drivers will be saved; the saved files will be restored when the **THEMISvme** package is removed.

The sample drivers contained in the package will be copied to the directory `/opt/THEMISvme/drv`. The user may choose to install these drivers by executing a script provided for that purpose. The sample drivers are not required for the normal operation of the SPARC 10MP system. -

2.2 Detailed Installation

During the course of installation, `pkgadd` will prompt the user for input on optional configuration. The first prompt will request the user to choose the packages to be installed:

The following packages are available:

```
1  THEMISvme  Themis SPARC10MP VME Drivers
      (sparc) 10MP 2.0
```

Select package(s) you wish to process (or 'all' to process all packages). (default: all) [`??.q`]:

Simply press <RETURN> to choose the THEMISvme package for installation.

The next input concerns the directory where the VME drivers will be placed:

```
Themis SPARC10MP VME Drivers
(sparc) 10MP 2.0
Themis Computer
```

Where should the driver objects be installed [`/kernel/drv`]
[`?.q`]

It is recommended that you choose the default input, `/kernel/drv`. If you choose to install the drivers in a directory other than `/kernel/drv` or `/usr/kernel/drv`, you may have to edit the `/etc/system` file to inform the kernel of the search path for driver object files.

The next prompt from `pkgadd` is:

The next prompt from pkgadd is:

```
This package contains scripts which will be executed with
super-user permission during the process of installing
this package
```

```
Do you want to continue with the installation of this
package [y,n,?]
```

The THEMISvme package contains custom scripts for easy installation of the package with minimal intervention from the user. These scripts need to be executed with super-user permission. Enter y to continue with the installation of the package.

After these prompts, pkgadd will proceed with copying the files to the system and installing the necessary drivers. At the end, you should see this message:

```
Installation of <THEMISvme> was successful.
```

If for any reason, the installation is not successful, please note down the error messages on the screen and contact Themis Technical Support.

2.3 Removing THEMISvme

If you want to remove the software interface of SPARC10MP for any reason you can execute the pkgrm command to remove the THEMISvme package:

```
# pkgrm THEMISvme
```

pkgrm will remove all the files contained in the package. The original contents of VME drivers before the package was installed will be restored.

2.4 *Installing THEMISvme On Diskless or Dataless Clients*

Currently, most of the software interface contained in THEMISvme can be installed on diskless and dataless client systems. On these clients, the VMEbus drivers will be copied and installed correctly. The header files and manual pages will be placed in the /opt/THEMISvme directory.

2.5 *Installing the Sample Drivers*

The /opt/THEMISvme/drv directory contains the source and binary versions of the sample device drivers in subdirectories 'src' and 'bin'. The subdirectory 'themis' includes header files that need to be copied to the /usr/include/themis directory on the system. The example programs rely on these header files.

To compile the drivers, you need to go into the subdirectory 'src' and type make. The compiled driver binaries will be placed in the 'bin' subdirectory.

To install the drivers, all the files in the 'bin' directory need to be placed in a directory that usually contains driver modules. (Typically this may be /kernel/drv or /usr/kernel/drv.) Then the drivers can be installed on the system by using the add_drv command.

Before using the drivers, you need to create files under the /dev directory. These files are symbolically linked to the actual device files under the /devices directory tree.

Scripts have been provided to perform the above tasks. The install sample scripts copies the drivers (as they are packaged) to the /kernel/drv directory. It then installs the drivers on the system, creates the files under /dev, and copies the header files into the /usr/include/themis directory. The remove sample script undoes all the installation done by the install script. You are strongly urged to carefully review the scripts before executing them.

If you need to make any changes to the drivers, you need to compile the drivers after the changes. After that you can copy the newly built driver object files to the /kernel/drv directory. You can install the new drivers on the running system by first executing the rem_drv command and then the add_drv command.

The remove sample script removes the sample drivers from the running system. The script deconfigures the drivers from the kernel and remove the driver object files from the /kernel/drv directory.

2.6 Sample Installation of THEMISvme

The contents of the screen during an installation of the THEMISvme package are given below. Input by the user appears in *italics*.

```
# pkgadd -d /dev/rmt/0
```

The following packages are available:

```
1  THEMISvme Themis SPARC10MP VME Drivers
    (sparc) 10MP 2.0
```

```
Select package(s) you wish to process (or 'all' to process
all packages). (default: all) {?,??,q}: all
```

```
Processing package instance <THEMISvme> from /dev/rmt/0
```

```
Themis SPARC10MP VME Drivers
(sparc) 10MP 2.0
Themis Computer
```

```
Where should the driver objects be installed [/kernel/drv]
{?,q} /kernel/drv
```

```
## Processing package information
```

```
## Processing system information
```

```
12 package pathnames are already properly installed.
```

```
## Verifying disk space requirements.
```

```
## Checking for conflicts with packages already installed
```

```
The following files are already installed on the system
and are being used by another package:
```

```
/kernel/drv/vme
/kernel/drv/vmemem
```

```
Do you want to install these conflicting files {y,n,?,q} y
```

This package contains scripts which will be executed with super-user permission during the process of installing this package.

Do you want to continue with the installation of this package [y,n,?] y

Installing Themis SPARC10MP VME Drivers as *THEMISvme*

```
## Executing preinstall script.
Saving current files .
[ verifying class <none> ]
## Installing part 1 of 1.
[ verifying class <devlink> ]
/kernel/drv/vme
/kernel/drv/vmedma
/kernel/drv/vmedma.conf
/kernel/drv/vmemem
[ verifying class <drv> ]
[ verifying class <include> ]
[ verifying class <manpage> ]
## Executing postinstall script.
Adding following devices to //etc/name_to_major
vmemem 67
Adding following devices to //etc/name_to_major
vmedma 95
Adding following permissions to //etc/minor_perm
vmemem:* 0666 bin bin
vmedma:* 0666 root sys
```

It will be necessary to do a reconfiguration reboot after driver installation. As soon as possible, execute 'reboot -- -r' to reboot. The new vme drivers will not be functional till a reboot is done.

Installation of *THEMISvme* was successful.

The following packages are available.

```
1  THEMISvmeThemis SPARC10MP VME Drivers
    (sparc) 10MP 2 0
```

Select package(s) you wish to process (or 'all' to process all packages). (default: all) [?,??,q]: q

2.7 Sample Installation of the Sample Drivers

The screen output when the install sample script is run on a system is included below (Characters typed by the user appear in italics)

```
# cd /opt/THEMISvme/drv
# ./install.sample
```

This script installs the sample device drivers for Solaris 2.4 to use the VMEbus interface on the Themis LXE+/564.

The script creates symbolic links under /dev for the device special files.

If you want to proceed, type y or Y. Otherwise, press any other key to abort

y

Replacing vmeintr interrupt driver

Replacing vmedvma DMA driver

Installing new drivers

Creating new devices

Copying header files

Installation completed.

```
-----
Installation Script for
Themis SPARC 10MP Solaris 2.4
sample device drivers
Version 1.2
-----
```


Using the VME Interface

3

On Themis systems, the power supply to the board is drawn through the VME chassis. If you are able to power the board up and run diagnostics, it is very likely that the basic VMEbus interface is functioning as intended. To verify the full functionality of the VMEbus interface and to illustrate the use of the sample device drivers, Themis provides a number of test programs. These are provided along with the source code.

This document gives a brief introduction to the different test programs provided by Themis Computer. Most of the test programs have a similar user interface. The reader may like to read the tutorial on the software interface provided by Themis to gain a fuller understanding of the concepts behind the test programs.

All the example programs (with the exception of the sample DMA program, work the same way on Solaris 1.x and Solaris 2.x systems.

3.1 Read/Write Test

This test performs simple read/write operations on VMEbus address space. The test requires the presence of memory space accessible over the VMEbus. Usually this space is provided by a memory card or through slave access to another VME board. Invoking the test program provides you with a command shell, from this shell you can execute any of the different tests.

A test session using the read/write test is given below. User input appears within angled brackets. The test system had a memory board on the VMEbus. The memory board was installed as an A32D32 board, with address range from 0x72a00000 to 0x749fffff.

```
# <./vme_rw>
```

```
Themis Computer - VME read/write test  program
```

```
Enter command, h for help : <h>
```

```
d adr[,length]  - display memory
```

```
h              - print this list
```

```
m adr[,value]  - modify memory
```

```
q              - quit program
```

```
z adr[,length] - zero memory
```

```
s mode         - set VME access mode
```

```
0x01 = A32D32, 0x02 = A32D16
```

```
0x11 = A24D32, 0x12 = A24D16
```

```
0x21 = A16D32, 0x22 = A16D16
```

```

Enter command, h for help : <s 1>
VME file /dev/vme32dj2 successfully opened

Enter command, h for help : <z 0x72a00000 0x1000>
Done.

```

```

Enter command, h for help : <d 0x72a00000 0x20>

```

```

0x72a00000: 00000000 00000000 00000000 00000000
0x72a00010: 00000000 00000000 00000000 00000000

```

```

Enter command, h for help : <m 0x72a00000 0x20>

```

```

0x72a00000: 00000000      ? <0>
0x72a00004: 00000000      ? <1>
0x72a00008: 00000000      ? <2>
0x72a0000c: 00000000      ? <3>
0x72a00010: 00000000      ? <4>
0x72a00014: 00000000      ? <5>
0x72a00018: 00000000      ? <6>
0x72a0001c: 00000000      ? <.>

```

```

Enter command, h for help : <d 0x72a00000 0x20>

```

```

0x72a00000: 00000000 00000001 00000002 00000003
0x72a00010: 00000004 00000005 00000006 00000000

```

```

Enter command, h for help : <d 0x72a00020 0x20>

```

```

0x72a00020: 00000000 00000000 00000000 00000000
0x72a00030: 00000000 00000000 00000000 00000000

```

```
Enter command, h for help : <d 0x74a00000 0x10>
```

```
0x74a00000:
```

```
Bus Error: received signal :10
```

```
Enter command, h for help : <d 0x73a00000 0x10>
```

```
0x73a00000: 00000000 00000000 00000000 00000000
```

```
Enter command, h for help : <q>
```

3.2 *mmap* Test

This test maps a chunk of VMEbus address space into user memory and performs a read or write operation on it. The test is very similar to the read/write test. However, instead of using `read()` and `write()` system calls, the test maps the specified region into user memory using `mmap`. This space can then be accessed like any other space in user memory.

The test requires the presence of memory space accessible over the VMEbus. Usually this space is provided by a memory card or through slave access to another VME board. Invoking the test program provides you with a command shell, from this shell you can execute any of the different tests.

A test session using the `mmap` test is given below. User input appears within angled brackets. The test system had a memory board on the VMEbus. The memory board was installed as an A32D32 board, with address range from 0x60000000 to 0x61ffff.

```
# <./vme_mmap>

Themis Computer - VME mmap test program

Enter command, h for help : <h>
d adrs[.length] - display memory
h              - print this list
m adrs[.value] - modify memory
q             - quit program
z adrs[.length] - zero memory
s mode        - set VME access mode
               0x01 = A32D32, 0x02 = A32D16
               0x11 = A24D32, 0x12 = A24D16
               0x21 = A16D32, 0x22 = A16D16

Enter command, h for help : <s 1>
VME file /dev/vme32d32 successfully opened.

Enter command, h for help : <z 0x60000000 0x1000>
Done

Enter command, h for help : <d 0x60000000 0x20>

0x60000000: 00000000 00000000 00000000 00000000
0x60000010: 00000000 00000000 00000000 00000000

Enter command, h for help : <m 0x60000000 0x20>

0x60000000: 00000000      ? <0>
0x60000004: 00000000      ? <1>
0x60000008: 00000000      ? <2>
0x6000000c: 00000000      ? <3>
0x60000010: 00000000      ? <4>
0x60000014: 00000000      ? <5>
0x60000018: 00000000      ? <6>
0x6000001c: 00000000      ? <.>
```

```
Enter command, h for help : <d 0x60000000 0x20>
```

```
0x60000000: 00000000 00000001 00000002 00000003 .  
0x60000010: 00000004 00000005 00000006 00000000
```

```
Enter command, h for help : <d 0x60000020 0x20>
```

```
0x60000020: 00000000 00000000 00000000 00000000  
0x60000030: 00000000 00000000 00000000 00000000
```

```
Enter command, h for help : <d 0x62000000 0x10>
```

```
0x62000000:  
Bus Error: received signal :10
```

```
Enter command, h for help : <d 0x61000000 0x10>
```

```
0x61000000: 00000000 00000000 00000000 00000000
```

```
Enter command, h for help : <q>
```

3.3 DVMA (slave mode access) test

Memory on the SPARC10MP board can be accessed from another board on the VMEbus chassis. This type of access is called slave mode access. Slave mode access under Solaris is tied to the concept of Direct Virtual Memory Access (DVMA). DVMA is a facility present on SPARC hardware that allows a device driver to specify virtual addresses rather than physical addresses for a DMA operation. The kernel maintains a map that specifies the correspondence between the DVMA address and the physical memory.

This section describes the many features of the software interface provided by Themis Computer. The features provide the system programmer with powerful ways of accessing the VMEbus from the Solaris Operating System environment. Themis Computer has developed many sample programs that illustrate the ways of using the software interface provided by Themis. These programs are available with the source code. These programs can be used to test the configuration of a VME system, they can also be used as example programs for the use of the different features provided by the software interface.

The following subsections introduce the concepts of accessing the VMEbus from user programs. Each concept is fully illustrated by a sample program. The reader is strongly encouraged to review the sample programs while reading this guide. The programs can be freely modified in any way that the user wants to. For more information on running the sample programs, please consult the document named *Using the VME Interface*. The reader may note that though the examples in this section use VMEbus memory boards, the usage can be extended to any type of VMEbus board.

The DVMA test allocates a DVMA region in memory and performs read/write operations on the region. The user interface of this test is quite similar to that of the mmap test except that instead of physical addresses, offsets into the DVMA region are used.

It is quite illustrative to use another system to perform slave mode access to the DVMA region. Usually the DVMA test is run on one machine, on another machine on the same VMEbus chassis, the mmap test is run. Both tests can then access the same region in memory.

In the following example, *slave* is the system that allocates a DVMA region, *master* is the system that performs a slave mode access to the region. The *slave-base-address* of the slave system is assumed to be 0x0, the master's *slave-base-address* needs to be such that the slave address regions of the two machines don't overlap. In this case the master's slave base address is 0xa00000.

On slave

```
# <./vme_dvma>
```

Themis Computer - VME DVMA test program

Enter command, h for help : <h>

```
d offset[,length] - display memory
h                  - print this list
m offset[,value]  - modify memory
q                  - quit program
z offset[,length] - zero memory
s size            - set up DVMA region
f                  - free DVMA region
```

Enter command, h for help : <s 0x2000>

id= 0x0, addr= 0x2ac8, sz= 0x2000

(The addr is used to access this region from another machine)

Enter command, h for help : <d 0 0x20>

```
0x00000000  9207bffc 4000779b f027bffc 81c7e008
0x00000010  91e80008 9de3bfad b4100019 92100018
```

On master:

```
# <./vme_mmap>
```

Themis Computer - VME test program

Enter command, h for help : <s 0x11>

VME file /dev/vme24d32 successfully opened.

Enter command, h for help : <d 0x2ac8 0x20>

0x00002ac8: 9207bffc 4000779b f027bffc 81c7e008
0x00002ad8: 91e80008 9de3bfa0 b4100019 92100018

(Note that the contents of the region are the same when
viewed
from different machines.)

Enter command, h for help : <z 0x2ac8 0x20>
Done

Enter command, h for help : <d 0x2ac8 0x20>

0x00002ac8: 00000000 00000000 00000000 00000000
0x00002ad8: 00000000 00000000 00000000 00000000

Now on slave.

Enter command, h for help : <d 0 0x20>

0x00000000: 00000000 00000000 00000000 00000000
0x00000010: 00000000 00000000 00000000 00000000

(Again, we can see that the two systems share the same view
of the contents of the region. Now we will modify the contents
starting from offset 0x20.)

Enter command, h for help : <m 0x20>

```
0x00000020: 0002e884      ? <1>
0x00000024: 0002e88c      ? <2>
0x00000028: 0002e898      ? <3>
0x0000002c: 0002e8a0      ? <4>
0x00000030: 0002e8a8      ? <5>
0x00000034: 0002e8b4      ? <6>
0x00000038: 0002e8c4      ? <7>
0x0000003c: 0002ea58      ? <8>
0x00000040: 0002ea5c      ? <9>
0x00000044: 0002ea68      ? <10>
0x00000048: 0002ea78      ? <11>
0x0000004c: 0002ea84      ? < >
```

On master:

Enter command, h for help : <d 0x2ae8 0x30>

```
0x00002ae8 00000001 00000002 00000003 00000004
0x00002af8 00000005 00000006 00000007 00000008
0x00002b08: 00000009 00000010 00000011 0002ea84
```

(We can see that the master can view the modified contents)

`dvmatest` allocates a small Direct Virtual Memory Access Region on the VMEBus address space and then writes the alphabetical characters into it. `vdma` is a general purpose program that allocates and frees DVMA regions. This test uses the `vmedvma` driver to set up a Direct Virtual Memory Access Region on the VMEBus address space. This memory can then be accessed either from the same machine or from another machine on the VME system.

`simulmaster` and `simulslave` are programs that illustrate the simultaneous access from multiple systems of a DVMA region allocated on one system. `simulslave` is run first, it allocates a DVMA region of 10 pages and outputs the region's size and virtual address. These are given as input to the `simulmaster` program on another system. The two programs try to access the same VMEBus address in alternating fashion. Suspending one of the programs will lead the other to stop, resuming the suspended program will make the other program start again.

3.4 Interrupts Test

This test requires the presence of two processor boards on the VME chassis. One of the boards is designated as the test initiator and the other as the test executor. Typically, a program on the test executor machine registers to process a particular interrupt. A program on test initiator machine is used to generate an interrupt on the VMEBus; this interrupt will be received by the test executor machine. The `vmeintr` driver should be installed on both the machines. The interrupts that can be received are specified by the configuration file for the `vmeintr` driver (usually `/kernel/drv/vmeintr.conf`).

Output from a test session is included below. In this case, the following interrupts were configured in the `vmeintr.conf` file: vector 0x80, priority 2 and vector 0x81, priority 3.

On test executor:

```
# <./vme_intr>
```

```
Themis Computer - VME interrupts test program
```

```
Enter command, h for help : <h>
```

```
e vector priority- enable an interrupt
```

```
s vector priority- send an interrupt
```

```
w - wait for an interrupt
```

```
d - disable (mask) an interrupt
```

```
h - print this list
```

```
q - quit program
```

```
Enter command, h for help : <s 0x80 2>
```

```
Set up to receive interrupt, vector : 0x80, priority. 2
```

```
Enter command, h for help : <w>
```

```
Waiting for interrupt. You have 30 seconds to generate it.
```

```
No interrupt received in the last 30 seconds
```

```
Enter command, h for help :
```

```
(The wait timed out after 30 seconds because no interrupt was  
generated.)
```

```
On test initiator
```

```
# <./vme_intr>
```

```
Themis Computer - VME interrupts test program
```

```
Enter command, h for help : <s 0x80 2>
```

```
Enter command, h for help
```

```
Now on test executor
```

```
Enter command, h for help : <w>
Waiting for interrupt. You have 30 seconds to generate it.
Received interrupt; vector 0x80, priority, 2
Enter command, h for help :
```

(You may see that the wait returns immediately because an interrupt was already pending.)

```
Enter command, h for help : <e 0x81 3>
An interrupt is enabled already. Disable it first
```

```
Enter command, h for help : <d>
Enter command, h for help : <e 0x81 3>
Set up to receive interrupt, vector : 0x81, priority, 3
Enter command, h for help : <q>
```

The roles of the test initiator and the test executor can be reversed at any time.

3.5 DMA Test

Direct Memory Access is the process by which a device takes control of the bus and performs data transfers to (and from) main memory or other devices. The device does this work without the help of the CPU. DMA transfers occur at significantly higher speeds as compared to normal transfers using `read()/write()` or `mmap()`.

On Themis SPARC10MP platforms, DMA access to the VMEbus is provided by the Fujitsu MBus to VMEbus Controller. The `vmedma` driver provides user access to the DMA features of the MVIC controller. Using the `ioctl` interface provided by the driver, the user program can utilize the DMA engine and achieve high rates of data transfer.

Typically a DMA transfer occurs between memory and the VMEbus. The memory location is specified by a user virtual address that is usually obtained via `malloc()` or `mmap()`. The VMEbus location is specified by a physical VMEbus address. The VMEbus address space does not have to be mapped into user memory.

In addition to transfers between VMEbus and memory, the `vmedma` driver also supports transfers from memory to memory and VMEbus to VMEbus. Depending on the capabilities of the hardware platform, these transfers may be emulated in software.

The `dmatest` program provides a way to test the DMA mechanism on the SPARC10MP product. To illustrate the high efficiency of DMA transfers, this program is controlled by command line arguments, unlike the other test programs which provide a command shell.

The usage of the program is

```
dmatest [ -hold ] [ -release ] [ -source <addr> ] [ -dest <addr> ] [ -size <count> ]
[ -loops <loop count> ] [ -blt ] [ -retain ]
[ -noblock ] [ -check ] [ -first ] [ -help ]
```

By default, the program performs a transfer from memory to memory, unless the `-source` and/or `-dest` options are specified. When `-source` or `-dest` is used, `<addr>` is a VMEbus address specifier as follows.

```
{a|64,32,24}|{d|64,32,16}|[<addrhi>|<addrlo>]
```

`<addrhi>` is only required when a64 addresses are used

To test VME to memory transfers, specify the VMEbus source address with the `-source <address>` option. To test memory to VME transfers, specify the `-dest <address>` options. For VME to VME transfers, specify both `-source` and `-dest`, and for memory to memory transfers, specify neither. The default address mode is A32D32.

The default transfer size is 0x10000 bytes, unless the `-size` option is specified. The default number of transfers is 10, unless the `-loops` option is specified.

Other options are

- | | |
|-----------|--|
| - size | - set size of each transfer |
| - loops | - set # of loops to run |
| - blt | - use block mode transfer |
| - retain | - use a retained buffer |
| - noblock | - run in non-blocking mode |
| - check | - check results every loop (not valid with -noblock, |
| - first | - only print first error in buffer |
| - help | - print the usage |
| - hold | - hold the DMA engine — no other args are valid |
| - release | - release the DMA engine — no other args are valid |

The -retain options request the program to lock the user addresses in memory. This options makes repeated transfers more efficient.

A test session using the dmatest is given below.

The test system had a memory board on the VMEbus. The memory board was installed as an A32D32 board, with address range from 0x72a00000 to 0x749fffff.

```
# ./dmatest -help
```

Usage:

```
./dmatest [-hold] [-release] [-source <addr>] [-dest <addr>]
          [-size <count>] [-loops <loop_count>] [-blt] [-retain]
          [-noblock] [-check] [-first] [-help]
```

When -source or -dest is used, <addr> is a VMEbus address specifier as follows:

```
[a{64,32,24}][d{64,32,16}:][<addrhi>.<addrlo>]
<addrhi> is only required when a64 addresses are used.
```

Examples

```
a32.50000000 - VMEbus A32 address 0x50000000
a24d16:500201f0 - VMEbus A32D16 address 0x500201f0
a64d32:10.500201f0 - VMEbus A64D32 address 0x10500201f0
```

Other options are:


```

-hold      - hold the DMA engine (no other options valid)
-release   - release the DMA engine (no other options valid)
-size      - set size of each transfer
-loops     - set # of loops to run
-blk       - use block mode transfer
-retain    - use a retained buffer
-noblock   - run in non-blocking mode
-check     - check results every loop (not valid with -noblock)
-help      - print the usage

```

```

#
# ./dmatest -dest 0x72a00000 -size 0x1000 -loops 100
Starting 100 loops
Done
Copied 400 000 KB in 0.180 seconds: 2222.222 KB/sec
#
# ./dmatest -dest 0x72a00000 -size 0x1000 -loops 100 -retain
Starting 100 loops
Done.
Releasing retained buffer (id fc252208)
Copied 400.000 KB in 0.092 seconds: 4347.826 KB/sec
#
# ./dmatest -source 0x72a00000 -loops 1000
Starting 1000 loops
Done
Copied 64000.000 KB in 10.179 seconds: 6287.455 KB/sec
#
# ./dmatest -source 0x72a00000 -loops 1000 -retain
Starting 1000 loops
Done
Releasing retained buffer (id fc252c30)
Copied 64000.000 KB in 8.368 seconds: 7648.184 KB/sec
#
# ./dmatest -source 0x72a00000 -dest 0x73a00000 -loops 1000 -retain
Starting 1000 loops
Done
Releasing retained buffer (id fc252750)
Copied 64000.000 KB in 23.509 seconds: 2722.362 KB/sec
#

```


Writing Programs for the VMEbus

4

The software interface of the 10MP is transparently implemented under Solaris 1.x (SunOS 4.1.3) and Solaris 2.x. The software interface is fully compatible with generic Sun4m VME architecture systems. Any VME device driver that conforms to the appropriate Solaris Device Driver Interface will run unmodified on Themis platforms.

This guide is intended for system programmers who are not familiar with the Solaris architecture and the concepts of the VMEbus. The guide discusses the concepts of Solaris drivers and VMEbus devices as they pertain to the Themis SPARC 10MP product. The guide is intended as a general introduction of the basic concepts of using the Themis SPARC 10MP product under Solaris. The guide is not intended as a reference document. The reader may refer to other documents for more detailed information. Convenient access to the SPARC 10MP User Manual and the on-line manual pages for Solaris would be helpful while reading this manual.

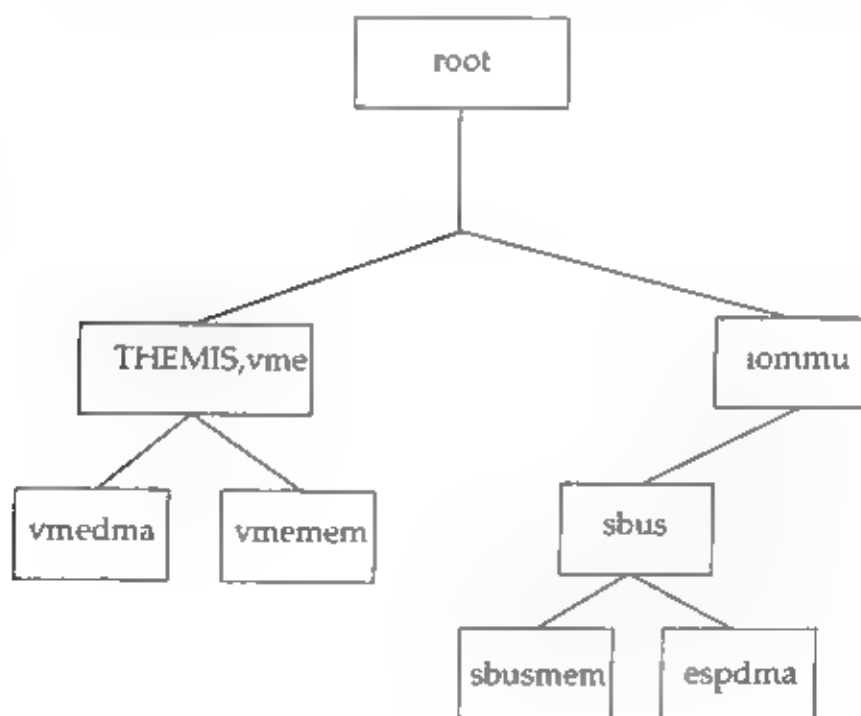
Solaris 1.1.1 (SunOS 4.1.3) is Sun's legacy operating system. The VME drivers for Solaris 1.1.1 are provided on tape by Themis Computer. The transparent interface enables the running of any off-the-shelf application on the SPARC 10MP. Solaris 2.x family is a new generation of operating system that unifies Sun's older BSD derived kernel and the newer UNIX SVR4 technology. The new operating systems add many new features like Symmetric Multiprocessing, Dynamically Loadable Drivers, etc.

In its current version, this guide focuses on the software interface provided under Solaris 2.x. However, except for section 4.1, the material in this guide applies to Solaris 1.1.1 systems also.

4.1 Solaris 2.x Device Hierarchy

Solaris 2.x systems support architecture independence of devices by using a layered approach. Each node in the tree structure has a specific device function. Standard devices are associated with leaf nodes. The drivers for these devices are called leaf drivers. The intermediate nodes in the tree are associated with buses, like SBus, VMEbus etc. These nodes are called bus nexus nodes and the drivers associated with them are called bus nexus drivers. The bus nexus driver encapsulates all the architectural dependencies of a particular bus. The leaf driver only needs to know the kind of bus it is connected to.

On Themis SPARC10MP systems, the device tree looks like this



In this diagram, `root`, `iommu`, `THEMIS`, `vme` and `sbus` are bus nexus drivers. The `iommu` encapsulates the features of the I/O Memory Management Unit found on sun4m systems. The `vme` driver encapsulates all the details of the implementation of the VMEbus interface. The `vmemem` driver, which is a leaf driver, only needs to know that it is connected to the VMEbus.

Solaris, like other UNIX systems, represents devices as special files in the file systems. These files are advertised by the device driver and are maintained by the `drvcontg(1M)` program. Usually the special files are created under the `/devices` directory and represent the hardware layout of a particular machine. `sysdet(1)` and `prtconf(1)` can be used to view the internal device tree. Symbolic links are created from the `/dev` directory to the special files in the `/devices` directory. User programs normally use the files from the `/dev` directory.

4.2 Features of the VMEbus

The VMEbus is an industry standard device interfacing bus that supports multiple address spaces. This means that a single VMEbus system can support devices that are 16-bit, 24-bit or 32-bit. There are many other features of the VMEbus which concern the system programmer. Another peripheral bus present in many SPARC platforms is the SBus.

VMEbus address space is not different from the physical memory space. A VMEbus address is indistinguishable from a memory address. Each card on the VMEbus has its own address. Usually this address is configurable. The address remains the same irrespective of the slot that the card is plugged into. In contrast, the SBus is geographically addressed, the address of a card is determined by the slot it is plugged into.

VMEbus supports multiple address spaces. A VMEbus system can support either 16, 24 or 32 address bits and either 16 or 32 data bits. Some VMEbus cards can respond to multiple size of address bits. These cards must be configured for the desired access mode. Usually this configuration can be done by jumper settings. It is necessary for the device driver developer to specify the address space supported by the device in the hardware configuration file.

VMEbus devices are not self-identifying; i.e. they do not provide information themselves to the system. Drivers for VMEbus devices need to have a `probe()` entry so that the kernel can determine if the device is really present. Additional information about the device must be provided in a hardware configuration file. See `driver.conf(4)` and `vme(4)` for more information.

VMEbus interrupts are vectored. When a device interrupts, it provides the priority as well as an interrupt vector. The kernel uses the information to uniquely identify the interrupt service routine to be called. The hardware configuration file must specify each priority-vector pair that the device may issue.

4.3 *Configuring the Software Interface of Themis SPARC 10MP*

The SPARC 10MP board can be accessed in a slave mode from another board on the VMEbus chassis. This type of access requires specifying the slave base address and the size of the slave access region of the 10MP board. Distinct values can be specified for 24 bit addresses and 32 bit addresses. On a VMEbus chassis with multiple boards, each processor board is required to have a non-overlapping region for slave mode access. The OBP attribute `vme32-slave-base` and `vme-boardid` can be used to set the slave base addresses of a 10MP board.

The OBP attribute `VME 32-Slave-base` specifies the A32 slave base address. The OBP attribute `vme-boardid` specifies the higher byte of the A24 slave base address. i.e. A value of 2 for `vme-boardid` specifies a value of 0x20 0000 for the A24 slave base address. Please refer to section 7 for details on modifying OBP attributes.

4.4 *Accessing the VMEbus from Solaris*

This section describes the many features of the software interface provided by Themis Computer. The features provide the system programmer with powerful ways of accessing the VMEbus from the Solaris Operating System environment. Themis Computer has developed many sample programs that illustrate the ways of using the software interface provided by Themis. These programs are available with the source code. These programs can be used to test the configuration of a VME system, they can also be used as example programs for the use of the different features provided by the software interface.

The following subsections introduce the concepts of accessing the VMEbus from user programs. Each concept is fully illustrated by a sample program. The reader is strongly encouraged to review the sample programs while reading this guide. The programs can be freely modified in any way that the user wants to. For more information on running the sample programs, please

consult the document named *Using the VME interface*. The reader may note that though the examples in this section use VMEbus memory boards, the usage can be extended to any type of VMEbus board.

4.4.1 Using Read/Write

The `vmemem` driver from Themis Computer enables the user to access any VMEbus address from a user program. Depending on the address space used by the particular VMEbus card, different devices need to be used.

Table 4-1 VME Table

File	Address Size	Data Transfer Size	Physical Address Range
<code>/dev/vme16d16</code>	16 bits	16 bits	0x0-0xFFFF
<code>/dev/vme24d16</code>	24 bits	16 bits	0x0-0xFFFFF
<code>/dev/vme32d16</code>	32 bits	16 bits	0x0-0xFFFFF
<code>/dev/vme16d32</code>	16 bits	32 bits	0x0-0xFFFF
<code>/dev/vme24d32</code>	24 bits	32 bits	0x0-0xFFFFF
<code>/dev/vme32d32</code>	32 bits	32 bits	0x0-0xFFFFF

e.g. to access a VME card that supports 24-bit addresses and 32-bit data transfers, the user needs to use the file `/dev/vme24d32`.

Using `/dev/vmeXXdYY` files is very similar to using normal files: the user program opens the file, uses the memory address as an offset to lseek to and does a read or write operation at the offset. The offset must fall within the physical address range of the file that was opened; also there should be a VMEbus device that would respond to the address.

To access 16 bytes at the VMEbus address 0xa0000, the following code snippet can be used

```
.
int fd;
int vme_base;
char buffer[16];

if ( ( fd = open( "/dev/vme24d32", O_RDWR) ) < 0)
{
    perror("In opening file /dev/vme24d32");
    return(-1);
}

vme_base = 0xa0000;
if ( lseek ( fd, vme_base, SEEK_SET) < 0)
{
    perror("In lseeking to 0xa000");
    close(fd);
    return(-1)
}

if ( read ( fd, buffer, sizeof(buffer) ) != sizeof(buffer) )
{
    perror("In reading 16 bytes");
    close(fd);
    return(-1);
}
.
```


In the above example, if there is no device on the VMEbus at address 0xa0000 the VMEbus operation will time-out and the program would receive a SIGBUS signal

The vmeXXdYY files can also be used to quickly check the proper installation of a VME board e.g. After installing a VME memory board at address 0x72000000, the user can use od(1) to look at the memory

```
> od -x /dev/vme32d32 +0x72000000
```

If the od command displays 0x72000000 and nothing else it indicates that there is no VME device at the address 0x72000000

Themis Computer provides the sample program vme_rw to illustrate the ways of accessing VMEbus addresses by using the vmeXXdYY files

4.4.2 Using mmap

A faster way to access VMEbus space is to use the mmap(2) system call. mmap() establishes a mapping between the user process's address space and a memory object represented by an open file. The /dev/vmeXXdYY files can be used for mmap() as well e.g. To access a region of length 1000 bytes in the VMEbus space, starting from location 0x500, the following code segment can be used

```
.
.
.
int vme;
caddr_t pageptr;

if ((vme = open("/dev/vme32d32",O_RDWR)) < 0) {
    perror("open error on VME file");
    return;
}
```

```
if ((pageptr = mmap((caddr_t)0L, roundup( 1000, PAGE_SIZE),
    PROT_READ|PROT_WRITE, MAP_SHARED, vme,
    (off_t)(0x500 & PAGEMASK ))) == (caddr_t)-1) {
    perror("mmap error");
    return;
```

Note that the 'off' parameter to `mmap()` is constrained to be aligned at a page boundary. The 'len' parameter does not have a size or alignment constraint. The success of `mmap()` does not imply that the specified range of VMEbus is valid and accessible. If the region specified by [off, off+len] is not a valid VMEbus address region, access to the region will result in a SIGBUS signal. It is often convenient to catch this signal and take some action. The following code snippet sets up a catching mechanism for SIGBUS and SIGSEGV.

```
.
void busError(int sig_num )
{
    printf("\nBus Error: received signal :%d\n", sig_num);
    return 0;
}

.
.
main()
.

    signal(SIGBUS, busError);
    signal(SIGSEGV, busError);
```

Themis Computer provides a sample program named `vme_mmap`. This program is a good example of using `mmap()` to handle regions of varying sizes. The program also handles bus errors generated by access to an invalid region. `vme_mmap` can also be used to test a VMEbus board in a number of ways.

4.4.3 DMA on the VMEbus

Direct Memory Access is the process by which a device takes control of the bus and performs data transfers to (and from) main memory or other devices. The device does this work without the help of the CPU. DMA transfers occur at significantly higher speeds as compared to normal transfers using `read()/write()` or `mmap()`.

On Themis SPRAC 10MP platforms, DMA access to the VMEbus is provided by the Newbridge SCV64 VMEbus Controller. The `vmedma` driver provides user access to the DMA features of the SCV64 controller. Using the `ioctl` interface provided by the driver, the user program can utilise the DMA engine and achieve high rates of data transfer.

Typically a DMA transfer occurs between memory and the VMEbus. The memory location is specified by a user virtual address that is usually obtained via `malloc()` or `mmap()`. The VMEbus location is specified by a physical VMEbus address. The VMEbus address space does not have to be mapped into user memory.

In addition to transfers between VMEbus and memory, the `vmedma` driver also supports transfers from memory to memory and VMEbus to VMEbus. Depending on the capabilities of the hardware platform, these transfers may be emulated in software.

The `ioctl` interface to the `vmedma` driver enables user programs to control the DMA transfer in a variety of ways. These include

1. Specify the VME address space for source and destination (16-bit, 24-bit, 32-bit or 64-bit).
2. Specify the size of data transfer (16-bit, 32-bit, or 64-bit)

3. Specify whether Block Transfer mode will be used (Block Transfer mode is a feature of VMEbus in which the address information is placed only once at the beginning of the data transfer. Subsequent data transfers in the same block cycle automatically increment the addresses.)
4. Specify that the user buffers be locked into memory and any setup work done in the kernel be retained. The buffers are later released by the `VIOCRELBUF` ioctl command. This feature greatly increases the efficiency of repeated DMA transfers.
5. Specify that the user program not block awaiting the completion of the DMA transfer. Completion of the transfer is indicated to the user process by the generation of a SIGIO signal.
6. Interrogate the capabilities of the underlying hardware mechanism, e.g. on Sparc 10MP, memory to memory DMA transfers cannot be performed in hardware. The driver will emulate such a transfer in software. The `VIOCGETCAP` ioctl command provides the user program to determine details such as these.

The following code segment illustrates a simple DMA transfer from memory to VMEbus address 0x72a00000

```
#include <themis/vmedmaio.h>

.

.

struct vme_dmacopy dma; /* DMA request structure */
int i;                  /* loop counter */
int fd;                 /* VME DMA engine fd */
long int * to_buf; /* memory buffer */
.
.

/* open the DMA engine */
if ((fd = open("/dev/vmedma0", O_RDWR)) < 0)
{
    (void) fprintf(stderr, "%s: can't open /dev/vmedma0: %s\n",
        argv[0], sys_errlist[errno]);
}
```

```

        exit(1);
    }

    /* allocate source memory buffer */
    if (!to_buf = (long int *)malloc ( 1024 * sizeof(long))) == NULL,
    {
        fprintf(stderr, "%s: can't allocate source memory buffer. %s\n",
            argv[0], sys_errlist[errno]);
        exit(1);
    }

    /* initialise the memory buffer with numbers from 0 to 1023 */
    for ( i = 0; i < 1024; i++)
        to_buf[i] = i;

    /* set up the DMA request
    *
    * - memory to VMEbus
    * - 32-bit data. 32-bit address
    * - no BLT
    * - do not retain buffers
    */

    dma.dma_source = to_buf, /* DMA transfer is from memory to VMEbus */
    dma.dma_dest = (void *)0x72a00000;
    dma.dma_sourceh1 = dma.dma_desth1 = 0; /* no 64-bit addresses */
    dma.dma_count = 1024 * sizeof(long), /* length of data transfer */
    dma.dma_result = NULL; /* we don't bother about the result area */

    dma.dma_flags = VF_DVME, /* destination is VME; no other special stuff */

    if ( ioctl(fd, VIOCDMACOPY, &dma) < 0) {
        fprintf(stderr, "%s: can't do DMA transfer: %s\n",
            argv[0], sys_errlist[errno]);
        free ( to_buf);
        exit(1);
    }

    free ( to_buf);

```

For VMEbus address spaces other than A32D32, the dma flags needs to be properly set. e.g for a DMA transfer from an A24D16 space to an A32D64 space, the flags are:

```
VF_SVME24 | VF_SVMED16 | VF_DVME | VF_DVMED64
```

Themis Computer provides sample test programs that illustrate the use of the vmedma driver

4.4.4 *Performing a Slave Mode Access to Another VME Board*

Memory on the SPARC10MP board can be accessed from another board on the VMEbus chassis. This type of access is called slave mode access. Slave mode access under Solaris is tied to the concept of Direct Virtual Memory Access (DVMA). DVMA is a facility present on SPARC hardware that allows a device driver to specify virtual addresses rather than physical addresses for a DMA operation. The kernel maintains a map that specifies the correspondence between the DVMA address and the physical memory.

In a typical operation, the master issues an address on the VMEbus, this address falls within the slave address range of another board on the chassis. The VME controller on the slave board then converts the VME address into a MBus virtual address. (For details on the conversion mechanism, refer to Section 6.2.5 of the SPARC10MP Technical Manual.) The VME controller then performs a DVMA operation on the MBus to access the physical memory location. The MBus virtual address generated by the VME slave access must be configured in the MVIC translation table. Themis Computer provides a driver that performs this configuration, the driver allocates a DVMA region and sets it up for VMEbus slave access at a particular VMEbus address. The master board simply uses this VMEbus address and generates a slave access operation on the VMEbus. The system that allocated the DVMA memory can also access the region by using special ioctl calls provided by the driver. Please refer to vmedvma(7) for more information on the driver.

The following code segment sets up a memory region of size 8192 bytes for DVMA access

```
#include <themis/vmedvma.h>

int fd;
struct vme_dvmacopy dma_req;

/* open the device */
if ((fd = open("/dev/vmedvma", O_RDWR)) < 0)
{
    perror("can't open /dev/vmedvma");
    exit(1);
}

/* allocate a DVMA region */
dma_req.size = 8192;
if (ioctl(fd, VDMA_ALLOCATE, &dma_req) != 0)
{
    perror("In allocating DVMA");
    close(fd);
    return(-1);
}

printf("Allocated a DVMA region id : %d, at virtual address : %x\n"
       dma_req.id, dma_req.addr);
```

The 'virtual address' output by this program can be used by other boards on the VMEbus chassis to gain access to this DVMA region e.g. If a second board on the same chassis needs to access this region, one can execute this command

```
> od -x /dev/vme24d32 [slave_address]
```

where [slave_address] is the virtual address output by the program

The DVMA region can be accessed from the same system by invoking the `ioctl()` calls implemented by the `vmedvma` driver. The following code segment fills a DVMA region with zeros, starting from offset 0x500

```
.
struct vme_dvmacopy dma_write,
char * buffer,

buffer = malloc( 4096);

dma_write.id = 1;
dma_write.addr= (caddr_t) buffer;
dma_write.size= 4096,
dma_write.offset= 2048,

if (ioctl(fd, VDMA_WRITE, &dma_write) != 0)
{
    perror("In initialising DVMA region");
    close(fd);
    return(-1);
}
.
```

Themis Computer provides the sample program `vme_dvma` that sets up a DVMA region for slave access. This region can then be accessed from another system on the VMEbus chassis using the `vmeXXdYY` files, the user can modify the contents of the region from one system and verify that both the systems share the same view of the contents of the region

4.4.5 *Advanced Features of the VMEbus Interface*

The SPARC10MP product provides powerful advanced features like generating a SYSRESET on the VMEbus, configuring the interrupt mechanism etc. Themis Computer periodically issues Technical Notes that detail these interfaces and ways to access them. These Technical Notes have been put together in the document named "Advanced Configuration of SPARC 10MP". Please contact Themis Technical Support for issues not covered in that document.

A device driver is a kernel module responsible for managing low-level I/O operations for a particular hardware device. Some device drivers manage 'fake' devices that exist only in software. A device driver contains all the device-specific code to communicate with a device. Like the system call interface protects application programs from the specifics of the platform, the device drivers protect the kernel from the specifics of the devices. A device driver also provides a standard I/O interface to the rest of the system, i.e., a user program should be able to open a device file and issue (in most cases a subset of) standard system calls to the file.

The VMEbus interface of Themis Computer's SPARC 10MP is transparently implemented under Solaris 1.x (SunOS 4.1.3) and Solaris 2.x. The software interface is fully compatible with generic Sun4m VME architecture systems. Any VME device driver that conforms to the appropriate Solaris Device Driver Interface will run unmodified on Themis platforms.

This guide is intended to highlight the VMEbus specific features that influence the design of drivers for VMEbus devices. It tries to provide basic information, and when needed, refers to sources for more detailed information on specific concepts. A full introduction to the principles of writing device drivers is beyond the scope of this guide. For information on writing device drivers for Solaris 1.x systems, please consult the document "Writing Device Drivers". The book "SunOS 5.3 Writing Device Drivers" has extensive information on writing device drivers for Solaris 2.x systems.

Themis Computer has developed a number of sample device drivers that illustrate the concepts covered in this chapter. Please see Chapter 8, *Sample Device Drivers*, for more information on the sample device drivers. These drivers, along with user level programs that interface to them, are provided in source code format.

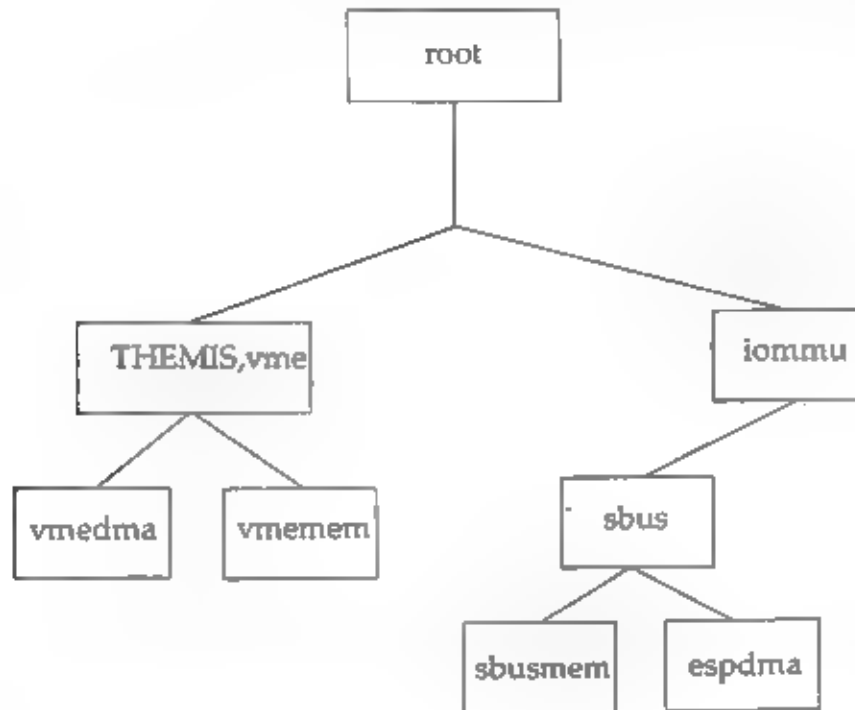
The last section in the guide discusses the situations where developing a custom device driver may not be necessary. The section discusses mechanism by which most device operations can be performed from the user level.

In its current version, this guide focuses on the software interface provided under Solaris 2.x. However except some sections, the concepts in this guide apply to Solaris 1.1.1 systems also.

5.1 *Solaris 2.x Device Hierarchy*

Solaris 2.x systems support architecture independence of devices by using a layered approach. Each node in the tree structure has a specific device function. Standard devices are associated with leaf nodes. The drivers for these devices are called leaf drivers. The intermediate nodes in the tree are associated with buses, like SBus, VMEbus etc. These nodes are called bus nexus nodes and the drivers associated with them are called bus nexus drivers. The bus nexus driver encapsulates all the architectural dependencies of a particular bus. The leaf driver only needs to know the kind of bus it is connected to.

On Themis SPARC10MP systems, the device tree looks like this.



In this diagram, root, iommu, vme and sbus are bus nexus drivers. The iommu encapsulates the features of the I/O Memory Management Unit found on sun4m systems. The vme driver encapsulates all the details of the implementation of the VMEbus interface e.g. the vmemem driver, which is a leaf driver, only needs to know that it is connected to the VMEbus.

Solaris, like other UNIX systems, represents devices as special files in the file systems. These files are advertised by the device driver and are maintained by the `drvconfig(1M)` program. Usually the special files are created under the `/devices` directory and represent the hardware layout of a particular machine. `sysdef(1)` and `priconf(1)` can be used to view the internal device tree. Symbolic links are created from the `/dev` directory to the special files in the `/devices` directory. User programs normally use the files from the `/dev` directory.

5.2 Features of the VMEbus

The VMEbus is an industry standard device interfacing bus that supports multiple address spaces. This means that a single VMEbus system can support devices that are 16-bit, 24-bit or 32-bit. There are many other features of the VMEbus which concern the system programmer. Another peripheral bus present in many SPARC platforms is the SBus.

VMEbus has many features that influence the design of device drivers written to operate on the bus. Some of these features require the VME board to be configured in a particular way. Some of the features require configuration from the Open Boot PROM. And some of the features need to be included in hardware configuration files for the device driver.

VMEbus address space is not different from the physical memory space. A VMEbus address is indistinguishable from a memory address. Each card on the VMEbus has its own address. Usually this address is configurable. The address remains the same irrespective of the slot that the card is plugged into. In contrast, the SBus is geographically addressed; the address of a card is determined by the slot it is plugged into.

VMEbus supports multiple address spaces. A VMEbus system can support either 16, 24 or 32 address bits and either 16 or 32 data bits. Some VMEbus cards can respond to multiple sizes of address bits. These cards must be configured for the desired access mode. Usually this configuration can be done by jumper settings. It is necessary for the device driver developer to specify the address space supported by the device in the hardware configuration file.

VMEbus devices are not self-identifying; i.e. they do not provide information themselves to the system. Drivers for VMEbus devices need to have a probe() entry so that the kernel can determine if the device is really present. Additional information about the device must be provided in a hardware configuration file. See `driver.conf(4)` and `vme(4)` for more information.

VMEbus interrupts are vectored. When a device interrupts, it provides the priority as well as an interrupt vector. The kernel uses the information to uniquely identify the interrupt service routine to be called. The hardware configuration file must specify each priority/vector pair that the device may issue.

5.3 Configuration Files for VME Device Drivers

Driver configuration files pass information about device drivers and their configuration to the system. Since VMEbus devices are not self-identifying, drivers for these devices need to use driver configuration files to inform the system that the device hardware may be present. The configuration file also must specify the device addresses on the VMEbus and any interrupt capabilities that the device may have. Please see `driver.conf(4)` and `vme(4)` for more details on the configuration files.

The syntax of an entry in the configuration file is:

```
name="driver name" class="vme" reg="      "
interrupts="      ".
```

Specifying class as "vme" indicates to the kernel that the device driver is for a VMEbus device and should be attached to the bus nexus driver for VMEbus. On SPARC10MP systems, the bus nexus driver is named `vme`.

Two common properties of interest to VME device drivers are the `reg` and `interrupts` properties. The `reg` property is an array of 3-tuples: address space, offset and length. Each 3-tuple describes a contiguous resource (registers, that can be mapped). The value of the address space is derived from the following table:

Table 5-1 Address Space

Address Space	Value
A16D16	0x2d
A24D16	0x3d
A32D16	0xd
A16D32	0x6d
A24D32	0x7d
A32D32	0x4d

e.g. For a device that supports A24D32 and has a set of registers that are 32-byte long and start at address 0xee00, the following 3-tuple can be used: 0x7d 0xee00,32. Multiple register sets are separated by commas.

VMEbus supports vectored interrupts. The interrupts property in the driver configuration file specifies all the interrupts that the device may generate. The interrupts property is an array of 2-tuples: VMEbus interrupt level, VMEbus vector number. e.g. for a device that generates interrupts at VME level 3 with a vector of 0x81, this two-tuple can be used: 2 0x81. The VMEbus interrupt level has a value between 1 and 7, and the vector a value between 0 and 255.

All VMEbus device drivers must provide reg properties. The first two integer elements of this property are used to construct the address part of the device name under /devices. Only devices that generate interrupts need to provide interrupts properties.

It is to be noted that the DDI/DKI functions are capable of handling the VMEbus specific features, provided that the hardware configuration file is correct. e.g. if the hardware configuration file correctly defines a register set, the ddi_map_regs() function can be used to map the register set into kernel address space, without any extra effort from the driver code.

5.4 Probing devices

Since VMF devices are not self-identifying, drivers for these devices are required to have a probe entry point. Usually the probe entry point tries to map the registers and then attempt to access the hardware using any of the peek and poke routines. The probe entry point should not initialize the device. In anyway, it also should not initialize any of the software state.

Mapping registers from VMEbus space does not need any special considerations. The ddi_map_regs() function call can be used to map the registers, this function will take care of the VMEbus address space that the register set is in.

The following code segment is an example of mapping a register set and 'carefully' accessing a byte in the set.

```

/*
 * xxprobe      - probe for this device
 */
static int
xxprobe(dev_info_t *dip)
{
    struct my_reg_str *regs; /* Device registers */
    caddr_t kaddr;           /* mapped register address */
    int retval;              /* return value */

    /*
     * If we're self-identifying, then we're here.
     */
    if (ddi_dev_is_sld(dip) == DDI_SUCCESS)
        return DDI_PROBE_SUCCESS;

    /*
     * Probe the device by mapping the registers, and reading the
     * command register and the parm7 register.. This assures that
     * "something" is there. The attach routine will try to run
     * more rigorous checks. So this simple check is all we
     * really need here.
     */
    if ((retval = ddi_map_regs(dip, 0, &kaddr, 0, 0)) != DDI_SUCCESS ||
        cmn_err(CE_WARN, "xx%d(probe) can't map my registers error%d\n",
                ddi_get_instance(dip), retval))
        return DDI_PROBE_FAILURE;
    }

    regs = (struct my_reg_str *)kaddr;
    retval = ddi_peekl(dip, (long *)&regs->command, (long *)0)
        == DDI_SUCCESS ? DDI_PROBE_SUCCESS : DDI_PROBE_FAILURE;

    if (retval == DDI_PROBE_SUCCESS)
        retval = ddi_peekl(dip, (long *)&regs->parm7, (long *)0)
            == DDI_SUCCESS ? DDI_PROBE_SUCCESS : DDI_PROBE_FAILURE;

    ddi_unmap_regs(dip, 0, &kaddr, 0, 0);

    return retval;
}

```

5.5 Registering Interrupts

VMEbus interrupts are vectored. When a device interrupts, it provides the priority as well as an interrupt vector. The kernel uses the information to uniquely identify the interrupt service routine to be called. The hardware configuration file must specify each priority-vector pair that the device may issue. Once this has been done, the driver code can call the `ddi_add_intr()` function to register the interrupts. On return from this function, the fourth argument contains a pointer to useful information like the VMEbus interrupt priority and the VMEbus vector number.

The following code segment provides an example of registering an interrupt with the kernel; the code first gets the number of interrupt specification in the configuration file, then it registers an interrupt handler for each interrupt specification.

```
static int
xxattach(dev_info_t * dip, ddi_attach_cmd_t cmd)
{
    struct xxstat * xsp;
    int nbint, i;

    if ( cmd != DDI_ATTACH)
        return (DDI_FAILURE);
    .
    .
    /* get the number of interrupt definitions */
    if (ddi_dev_nintrs(devi, &nbint) == DDI_FAILURE)
    {
        cmd_err(CE_WARN, "xx: No interrupts property.");
        return (DDI_FAILURE);
    }

    for ( i=0, i < nbint; i++)
    {
        /* Register first with a null handler so that the interrupt
         * routine doesn't grab the mutex
         */
        if ( ddi_add_intr(dip, i, &xsp->iblock_cookie[i], NULL,
```



```

        ( (u_int *) (caddr_t) nulldev, NULL) != DDI_SUCCESS

    {
        cmn_err(CE_WARN, "xx: cannot register interrupt.");
        return (DDI_FAILURE);
    }

    /* Initialise the mutex now. */
    mutex_init ( &xsp->mu, "xx mutex", MUTEX_DRIVER,
        (void *) xsp->iblock_cookie[i]);

    /* Remove the interrupt and register again with the correct
     * interrupt handler.
     */
    ddi_remove_intr ( dip, i, xsp->iblock_cookie[i]);
    if ( ddi_add_intr(dip, i, &xsp->iblock_cookie[i],
        &xsp->idevice_cookie[i], xxintr, NULL) != DDI_SUCCESS

    {
        cmn_err(CE_WARN, "xx: cannot register interrupt ",
            return (DDI_FAILURE);
    }

    cmn_err (CE_NOTE, "Registered interrupt %d with priority %d and
        vector 0x%x",
        i, xsp->idevice_cookie[i].idev_priority,
        xsp->idevice_cookie[i].idev_vector),
}

```

Themis Computer provides a sample device driver `vmeintr` to illustrate the use of interrupts in device drivers. This driver is provided along with the source code and a sample configuration file. The reader may wish to study the source code for this driver to gain a better understanding of handling interrupts in device drivers.

5.6 *Allocating DVMA Space*

Direct Virtual Memory Access is a facility present on SPARC hardware that allows a device to specify virtual addresses rather than physical addresses for a DMA operation.

The kernel maintains a map that specifies the correspondence between the DVMA address and the physical memory. It is the responsibility of the device driver to set up the memory region for DVMA access.

Setting up a DVMA region is a two-step process: allocating memory and allocating DMA resources for this memory region. The following code segment is an example of setting up a DVMA region of size 'len' bytes.

While allocating memory and setting it up for DVMA, the driver can describe the capabilities of the DMA engine to the kernel by using a `ddi_dma_lim_t` structure. On SPARC IOMP systems, the capabilities are as follows:

```
static ddi_dma_lim_t limits =
{
    0x0,
    0xffffffff,
    0xffffffff,
    0x1,
    0x1,
    0,
    1,
};

caddr_t kmem_base;
int req_id, real_len, req_len;
ddi_dma_handle_t handle;
ddi_dma_cookie_t cookie;
dma_req_t *reqp;

req_len = len;
real_len = 0;

/* allocate memory */
```

```

        if (ddi_mem_alloc (xsp->dip, &limits,
                           (size_t)req_len, 0, &kmem_base,
                           (uint *) &real_len) != DDI_SUCCESS)
        {
            cmn_err (CE_NOTE, "ddi_mem_alloc failed.");
            return (ENOMEM);
        }

/* set it up for DMA access */
        if (ddi_dma_addr_setup (xsp->dip, (struct as *) NULL,
                                kmem_base, real_len,
                                DDI_DMA_RDWR, DDI_DMA_DONTWAIT, 0, &limits,
                                &handle) != DDI_DMA_MAPPED)
        {
            ddi_mem_free (kmem_base);
            cmn_err (CE_NOTE, "ddi_dma_addr_setup failed.");
            return (EFAULT);
        }

/* extract the virtual address of the memory region.
 * this can be obtained from the DMA cookie structure.
 */

        if (ddi_dma_htoc (handle, 0, &cookie) != DDI_SUCCESS
            ||
            ddi_dma_sync (handle, 0, real_len, DDI_DMA_SYNC_FORDEV,
                          != DDI_SUCCESS))
        {
            ddi_dma_free (handle);
            ddi_mem_free (kmem_base);
            cmn_err (CE_NOTE, " ddi_dma_sync() failed\n");
            return (EFAULT);
        }

        cmn_err ( CE_NOTE, "Virtual address of memory  is 0x%x", kmem_base ;
        cmn_err ( CE_NOTE, "Virtual address of the DVMA region is 0x%x",
                   (caddr_t) cookie.dmac_address );

```

The address contained in the 'DMA cookie' is the 'virtual address' of the DVMA memory region and can be used to access the memory region over the VMEbus e.g. if the 'virtual address' printed by the second `cmn_err` statement is 0x588, you can access this memory region from another system on the VMEbus chassis by using this command

```
# od -x /dev/vme24d32 0x588
```

Themis Computer provides a sample device driver `vmedvma` to illustrate the use of DVMA in VME device drivers. This driver is provided along with the source code and a sample configuration file. The reader may wish to study the source code for this driver to gain a better understanding of using DVMA from device drivers.

5.7 Mapping VMEbus Space

On some occasions, it would be convenient to map a chunk of VMEbus address space into the system's memory. From the user level, this can be done by doing a `mmap()` operation on the appropriate `/dev/vmeXXdYY` file. From the driver level, mapping VMEbus address space can be done by using the `ddi_map_regs()` function e.g. to map two pages of VMEbus space starting at location 0x66600000, the following code segment can be used

```
if (ddi_map_regs(dip, 0, &map_space, 0x66000000, off_t, 8192) ==  
    DDI_SUCCESS)  
{  
    return (EFAULT);  
}
```

On return from the function, `map_space` contains the base address of the kernel virtual region.

There are many caveats to using `ddi_map_regs()` to map a portion of VMEbus address space

1. An A32D32 device can only map a chunk from A32D32 address space

2. The address space indicated by [offset, offset+len] must be a valid region on the VMEbus.
3. Access to the mapped region should be localized. Subsequent access to locations that are far apart will result in many page faults, particularly on regions of larger sizes.

It is prudent to use the `ddi_peek()` and `ddi_poke()` routines to access a region mapped using `ddi_map_regs()`.

5.8 *Driving Devices Without Writing Device Drivers*

As mentioned before, device drivers protect the rest of the kernel from the specifics of hardware devices. Most hardware devices would necessitate the development of a device driver before they can be used on a computer system. However, the features of the VMEbus and the design and implementation of the *Themis* Computer software interface for VME alleviate the need for specialised device drivers for most simple VME devices. This provides a tremendous advantage to programmers writing applications for these simple devices.

In *Solaris 2.x* device drivers are dynamically loadable. This avoids the need to relink the kernel and reboot the system whenever a driver is modified. However, a fault in the driver could still cause the kernel to panic, resulting in long downtimes for the machine. Further, an abnormal shutdown of the machine, as in the case of kernel panic, could cause the disk to be corrupted and some important files to be lost. These problems can be avoided if the program driving the device executes in user mode instead of kernel mode.

This section discusses some ways by which a user program can do device operations that are traditionally performed by a device driver. The section is intended to clue the programmer on to the possibilities of accessing devices from user level. The reader is encouraged to go through the source code of the sample drivers to gain an insight into how they can be used for user level access.

In simplistic terms, a device can be viewed as a collection of registers that are addressable. Almost all of these registers can be either read from or written to. Some of them allow both read and write access. Access to many of the registers results in effects like resetting the value of the register, clearing the interrupt etc. Some registers require a particular sequence of accessing them. Some registers have specific data widths that can be used to access them. Though

there are some devices that have some very peculiar behavior, most devices can be modeled as a collection of registers. Such devices, if they are VMEbus devices, can be programmed and used without the need for a specialised device driver.

As mentioned before, VMEbus addresses are not distinguishable from memory addresses. The `vmemem` drivers from Themis Computer, which is the device driver for the `/dev/vmexxdyy` files, enables an user program to access any address in the VMEbus space.

This feature can be used to access the registers of a device from a user program. The user program needs to open the appropriate `/dev/vmeXXdYY` file, lseek to the address of the device registers and do a read or write operation. The `vmemem` driver performs the read/write operation carefully by using the `ddi_peek()` and `ddi_poke()` calls. If the user program opens the appropriate file and the read/write access meets the alignment criteria of the device, the device registers can be easily programmed from the user program.

e.g. To access a device that supports 32 address bits and has 32-bit data, with the registers starting at offset `0x7e000000`, the user program needs to open `/dev/vme32d32` and lseek to offset `0x7e000000`. Please note that the registers of this device could be 8-bit wide, 16-bit wide or 32-bit wide. If the read/write call is issued with the appropriate length, the `vmemem` driver will issue the appropriate `ddi_peek` or `ddi_poke` call.

Note that the `mmap()` interface of the `vmemem` driver does not use the `ddi_peek/ddi_poke` calls. In such cases, access to all non-existent address space will cause a SIGBUS signal to be sent to the user program.

The `vme DVMA` driver provided by Themis can be used to support more powerful devices that are capable of performing DMA operations on the VMEbus. e.g. To program a device to perform a DMA operation to on-board memory space, the user program can do the following:

1. Open the relevant `/dev/vmeXXdyv` file and lseek to the beginning of the register space.
2. Open `/dev/vmedvma` and allocate a DVMA region.
3. Use the virtual address returned by the `allocate_ioctl` to program the DMA engine on the device.
4. Use the other `uxctls` provided by the `vmedvma` driver to access the DVMA region.

Devices that issue interrupts pose more of a problem when accessed from the user level. In some cases the devices can be programmed to not raise interrupts. Input/output in such cases may be done by making the user program 'busy wait'. In other cases, the `vmeintr` driver can be used to receive the interrupt and signal a user program when the interrupt is raised. The user program can then look at the device registers and take appropriate actions.

Future releases from Themis Computer will include a "user level driver" that can be used to program a simple device.

6.1 Introduction

This guide is intended to provide helpful information and tips to systems programmers who use the Themis SPARC 10MP boards. The next section is intended to address the programmers who wish to program the Themis SPARC 10MP board for the VMEbus without the aid of an operating system or real time kernel. Further sections are intended to address the programmers who wish to access the Themis SPARC 10MP board from any of the supported Operating System environments.

Detailed information concerning on-board device interfaces, memory maps and device accesses are contained in the appendices and should be consulted while reading this guide.

The Themis SPARC 10MP processor board implements a Sun SPARCstation10 compatible workstation on a single or double width 6U VMEbus board. A full VME master/slave interface is included, using the Fujitsu MBus to VMEbus Interface Controller.

This guide is primarily concerned with the implementation and programming of 10MP that are not common to the superSPARC family of workstations (SPARCstation 10).

6.2 Overview of VME Interface Under Solaris 1.1, 1B/2.X

Great effort has been expended to achieve complete compatibility between Sun VME implementations and the device driver interface implemented for the IOMP board under Solaris.

This section is primarily concerned with the implementation and programming of IOMP specific features that are not common to the microSPARC II family of workstations (SPARCstation 5). It assumes that the reader is familiar with the sun4m architecture and the concepts of VMEbus and Device drivers. A separate tutorial provides more detailed information for readers who are not familiar with these concepts.

The VMEbus interface of the IOMP is transparently implemented under Solaris 1.X (SunOS 4.1.3) and Solaris 2.X. The software interface is fully compatible with generic Sun4m VME architecture systems. Any VME device driver that conforms to the appropriate Solaris Device Driver Interface will run unmodified on Themis platforms.

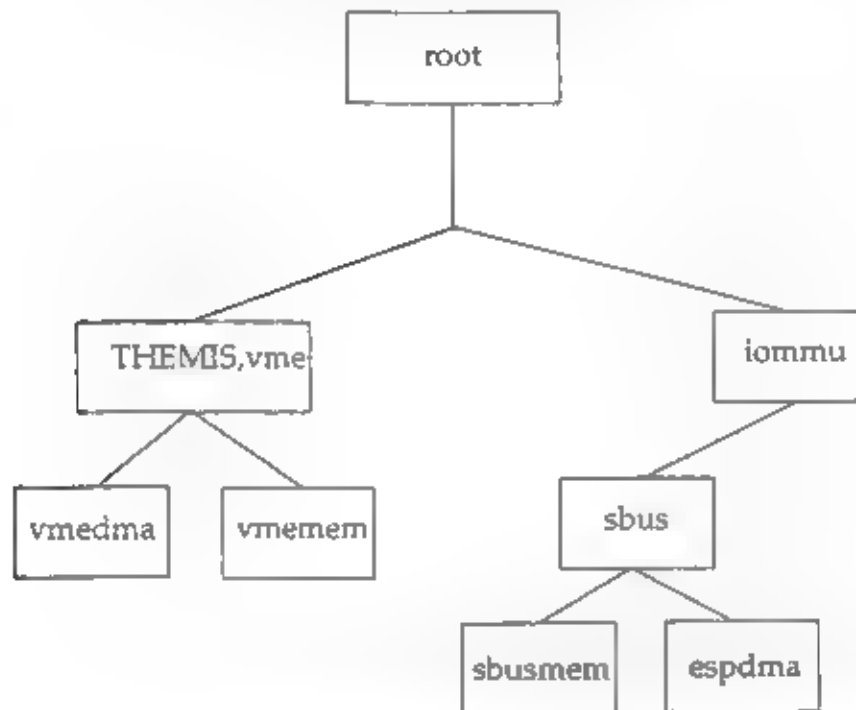
Themis Computer has developed a number of device drivers to provide the necessary software interface needed by system programmers. The drivers can be classified into three categories.

- VMEbus Nexus drivers
- Utility drivers
- Sample drivers

The drivers are described in the following subsections. Themis also provides online manual pages for all the drivers included in the software interface. These manual pages include more detailed information on the drivers.

6.2.1 VMEbus Nexus Driver

The `tvme` driver forms the core of the software interface provided by Themis Computer. `tvme` is a bus nexus driver that encapsulates all the architectural features of supporting the Themis 10MP board on Solaris systems. This driver is configured to be permanently loaded as the bus nexus driver for the VMEbus.



Any driver that specifies its class or parent as `vme` will be attached to the `vme` driver. The `vme` driver provides support for all VMEbus operations, including VME interrupt processing, Direct Virtual Memory Access (DVMA) on the VMEbus, mapping of registers on the VMEbus to kernel space, mapping of VMEbus memory to user address space through `mmap()`, etc.

The `ivme` nexus driver also exports certain capabilities that can be used by VMEbus leaf drivers. This enables leaf drivers to perform operations like Direct Memory Access in a very efficient manner.

The `ivme` driver recognizes the standard configuration file for VME device drivers (see `vme(4)`). The configuration of any `vme` device driver specifies the VMEbus interrupt level and the interrupt vector. The mapping of the VMEbus interrupt level to the SPARC interrupt priority level is as follows:

Table 6-1 VMEbus Interrupt Mapping

VMEbus		1	2	3	4	5	6	7
SPARC ip	1	2	3	5	7	9	11	13

6.2.2 Utility Drivers

These drivers are a standard part of the software interface provided by `Thema.s` Computer. These drivers provide a useful interface to system programmers who wish to utilize the different features of the VMEbus architecture.

The `vmemem` device driver provides the standard `/dev` devices for accessing the VMEbus from user processes.

Table 6-2 VME Device Drivers

Device File	Address Size	Data Transfer Size
<code>/dev/vme32d32</code>	32	32
<code>/dev/vme32d16</code>	32	16
<code>/dev/vme24d32</code>	24	32
<code>/dev/vme24d16</code>	24	16
<code>/dev/vme16d32</code>	16	32
<code>/dev/vme16d16</code>	16	16

`Read()`, `write()` and `mmap()` calls are supported and no special `ioctl()` calls are required to configure the interface. VME interface configuration is performed under OBP and should not be modified while the operating system is running.

On Themis SPARC 10MP platforms, DMA access to the VMEbus is provided by the Fujitsu MVIC. The `vmedma` driver provides user access to the DMA features of the MVIC. Using the `ioctl` interface provided by the driver, the user program can utilise the DMA engine and achieve high rates of data transfer.

6.2.3 Sample Drivers

The software interface provided for the 10MP platform fully supports standard VMEbus device drivers written for Solaris environments. It is the intention of Themis Computer to fully support users who wish to write their own VME device drivers that would function on Themis SPARC 10MP platforms. To aid these users and to illustrate the specific features of the VMEbus architecture, Themis provides a number of sample device drivers. Themis provides the complete source code and the necessary configuration files for these drivers.

- `vmemtr` is a sample driver provided by Themis Computer to illustrate the vectored interrupt mechanism of the VMEbus. The `vmemtr` driver relies on the driver configuration files to specify the interrupts it should handle. Each instance of the driver registers the interrupts with the system. Through the `ioctls` implemented by the driver, the user can generate interrupts and send them to the appropriate driver instances. The interrupt generator and the interrupt receiver need to run on different computers.
- `vmedma` is a sample driver provided by Themis Computer to illustrate the use of Direct Virtual Memory Access within a device driver. The user can ask the driver to allocate a DMA region on the VMEbus. The driver allocates all the necessary resources to support a DVMA transfer to this region. The driver also implements mechanisms by which the user program can write to or read from the DVMA region.
- `simplifiedma` is a sample driver provided by Themis Computer to illustrate the use of Direct Memory Access within a device driver. It is a simplified version of the standard `vmedma` driver. This driver is mainly intended for programmers writing device drivers that perform DMA operations on the VMEbus.

The SPARC 10MP product provides powerful advanced features like generating a SYSRESET on the VMEbus, configuring the interrupt mechanism etc. Themis Computer periodically issues Technical Notes that detail these interfaces and ways to access them. These Technical Notes have been put together in this document. Please contact Themis Computer Technical Support for advanced configuration topics not covered in this document.

Some of the sections in this chapter may apply to other Themis Computer products too. These are noted as applicable.

Problem By default, VME interrupts are enabled on THEMIS boards. This can cause conflicts on the VMEbus in multi CPU configuration, as there may be more than one interrupt handler for one particular VME interrupt level.

Programming of VME interrupt mask is done under OBP, using nvramrc commands

```
ok nvedit
1 xx 5ffe 0014 20 spacec'
2: ^C
ok nvstore
ok setenv use-nvramrc? true
ok reset
```

- bit [7 1] Vme IRQ 1 7 If set IRQ is enabled
- bit 0 Arbitration mode 0 = Single level, 1 = Round robin

Rev Date 3/20/96

THFMIS LXE+ and 5-64

```

-----
ok nvedit
1 xx 7d00.00b0 20 space!
2 ^C
ok nvstore
ok setenv use-nvramrc? true
ok reset

```

VME BUS interrupt mask register description is as follows.

- bit 6: VME IRQ7
- bit 0: VME IRQ1

(*) On the LXE running solaris 1 x, you need to disable interrupts from UNIX, after the system is booted. Please contact Technical support for details.

7.2 Isolating Boards from VMEbus SYSRESET

TECHNICAL NOTE No 120-00
SUBJECT: Isolate board from VME SYSRESET
BOARD: THEMIS LXEplus, 5-64, 10MP
O/S: all
Hardware rev: all
DATE: May 9, 1995

It can be useful to isolate a VME board from the VME sysreset signal, especially if you are running Solaris.

To isolate the Themis 5-64, or the LXEplus from the VME sysreset, you have to remove a CMS resistor (R5500). After removing that resistor which is in fact a short (0 Ohms), the board will be disconnected from the VME sysreset signal. In case you want to connect the board to the sysreset signal again, use a solder bead to short R5500.

NB The location of R5500 is different on the 5-64 and the LXEplus. Please see drawings enclosed.

To isolate the 10MP from SYSRESET, you need to change the socketed PAL U2309 with a special PAL. Please contact Technical support to get that PAL. Note that after that modification, the 10MP won't receive SYSRESET from other boards, but will still be able to send it on the bus.

7.3 Handling Data Mismatch Over the VMEbus

TECHNICAL NOTE No 133-00
SUBJECT Data mismatch over the VME
BOARD THEMIS 10MP
O.S Solaris 2.4
Hardware rev all
DATE July 1, 1995

There is a bug in solaris 2.4, in the way the MicroSparcII MMU is handled. This bug can result in data mismatch over the VME. Typically, you won't read back the same value that you write to a given VME memory range, especially if you make transfers of more than one MMU page (4k).

The solution is to apply the Sun 101945 kernel jumbo patch. This is quite a big patch (16MB), and Themis is shipping it along with the 10MP VME nexus driver.

To apply it, you simply need to run the `/installpatch` command at the patch directory level.

```
# cd 101945-27
# ./installpatch
```

To check if you already installed that patch, you need to run the following command:

```
# showrev -p
```

This will tell you what patches were already installed on your system.

7.4 Handling VMEbus Lock-Up Problems

TECHNICAL NOTE No 137-00
SUBJECT: VME lock-up problem on the 10MP
BOARD: All
O.S.: all
Hardware rev: all with C4 KSIC (and before)
DATE: October 18, 1995
STATUS

There is a potential problem on the 10MP that after heavy SBUS and VME traffic, the board can no longer access VME

The problem is that the VMMU in charge of translating SBUS addresses to VME addresses, is disabled at random times. Thus no further VME access is possible, and the CPU gets a bus error that will crash the system if the access is made by kernel code (device driver).

The solution is to use the C5 KSIC (and above versions). This KSIC is actually disabling accesses to the VMMU registers, preventing the VMMU to get disabled.

Note: How to check the version of your KSIC:

-> Please check the revision number that is written onto the KSIC itself (Right in the middle of the 2 SBUS connectors)

7.5 Configuring VMEbus Release Mode

TECHNICAL NOTE No 122-00
SUBJECT VME release mode ROR vs RWD
BOARD THE VIS LXE,plus,10MP
O/S all
Hardware rev: all
DATE Oct 19, 1995

When a MASTER VME board wants to make a VME BUS cycle, it has to request the VMEBUS from the bus controller (SYSCON). Then when the cycle is done, it will release the VMEbus, so that other MASTERS on the VME can request and be granted the VMEBUS.

There are 2 modes of releasing the VMEbus

- ROR: Release on request.
- RWD: Release when done.

-In ROR mode the MASTER VME board will keep ownership of the VMEBUS (i.e. keep BBSY asserted), until another MASTER request the VMEBUS. But one should be aware that this could result in VMEBUS starvation for every board located to the left of any ROR master. This is due to the fact that the SYSCON is granting ownership of the VMEBUS using the BUS GRANT daisy chain signals. Those signals are propagated from left to right of the chassis, starting from the SYSCON to the next VME board (board #1). If that next board is requesting the VMEBUS then it will consider it has been granted the VMEBUS by the SYSCON.

Otherwise, it will pass the BUS GRANT daisy chain signal to the next board on its right (board #2).

Now if board #1 is programmed in ROR mode,

-In RWD mode, the MASTER VME board releases the VMEBUS as soon as the VME cycle it just performed is done.

How to program your board to switch to coupled mode

Therms 10MP and LXFplus:

=====

The default mode is RWD. Here is how to set the board in ROR mode.

You have to use OBP nvedit feature to add a FORTH command, that will be ran each time the board is booted

```
ok nvedit
1 7d00.0090 20 spacel@ df and 7d00.0090 20 spacel'
2: ^C
ok nvstore
ok setenv use-nvramrc? true
ok reset
```

4. Edit /etc/hosts and add an entry for the client:

```
#
# Sun Host Database
#
# If the NIS is running, this file is only consulted when booting
#
127.0.0.1          localhost
#
198.211.242.189 mp112 loghost
198.211.242.188 564_1
~
" /etc/hosts" 9 lines, 167 characters
```

5. Same thing under /etc/ethers (tftpboot)

```
0:80:b6.1 2 3      564_1
~
" etc/ethers" [New file] 1 line, 20 characters
```

6. Run add_client:

```
mp112# /usr/etc/install/add_client -i
```

```

CLIENT FORM          [?=help] [DEL=erase one char] [RET=end of
                        input data]
-----
Architecture Type : x[sun4m sunos.4.1.4]
Client name       : 564_1
Choice            : x[create] [delete] [display] [edit]
Root fs : /export/root (sd0a)          144616448   Hog : sd0h   0
Swap fs : /export/swap (sd0a)          144616448   Hog : sd0h   0

Client Information :
    Internet Address      : 198.211.242.188
    Ethernet Address      : 0:80:b6:1:2:3
    NIS Type               : x[none] [client]

    Swap size (e.g. 8B, 8K, 8M) : 16M
    Path to Root              : /export/root/564_1
    Path to Swap              : /export/swap/564_1
    Path to Executables       : /usr
    Path to Kernel Executables : /export/exec/kvm/sun4m sunos 4 1 4
    Path to Home              : /home/mp112
    Terminal type             : sun

Ok to use these values [y/n] ?
    [x/X=select choice] [space=next choice] [^B/^P=backward]
    [^F ^N=forward]

```


7.6 Installing Solaris 1.x Patches on Diskless Clients

TECHNICAL NOTE No 145-00
SUBJECT: Homogeneous server Client install problem.
BOARD: THEMIS LXEplus,10MP
O.S.: Solaris 1
Hardware rev: all
DATE: Nov, 1995

Themis VME boards are sometimes configured as a client diskless node. They use httpboot, to boot from a Sun server machine. Then they mount their /root, /swap, /usr and /home file systems from a remote disk, using NFS. That kind of configuration can sometimes cause a problem, when installing the VME patch, that will permit the Themis board to access the VME bus.

Under Solaris 1, VME software is provided under the form of modified Kernel binaries, not true device drivers like under solaris 2.4. So, applying the VME patch on the NFS mounted partitions of the client, may sometimes result in patching some important kernel binaries on the server itself.

The VME patch is only updating files located in the /usr/kvm directory, also known as the 'kernel binaries'. So, it is of the utmost importance to configure the client with a separate /usr/kvm directory than that of the server. In the case, where the diskless and the server are running 2 different versions of Solaris 1 (let's say solaris 1.1.2 and Solaris 1.1.1B), there is no problem, as in this case, running "add_services" will generate a /usr/kvm (kernel binaries) unique to the the client diskless. This directory will actually be located in /export/exec/sun4m.sunos.4.1.1.

But in the case where the client and the server are running the exact same version of the Solaris 1 Operating system, "add_services" won't let you install a separate /usr/kvm for the client. Themis is currently in contact with SunSoft, to find a solution to that problem.

In the meantime, there is a workaround to install, let's say a Solaris 1.1.2 diskless client on a Solaris 1.1.2 server machine.

(in the example below, mp112 is the server, and 564_1 is the client.)

1. During the Sysgen of the server, don't forget to configure that machine as a server (versus a stand alone system).
2. Under /export/exec/kvm/ directory, remove the sun4m.sunos.4.1.4 directory (actually a link to /usr/kvm).
3. Copy /usr/kvm in /export/exec/kvm/sun4m.sunos.4.1.4.

```
mp112# cd /usr/kvm
mp112# tar cvf /tmp/kvm
mp112# cd /export/exec/kvm
mp112# mkdir sun4m.sunos.4.1.4
mp112# cd /export/exec/kvm/sun4m.sunos.4.1.4
mp112# tar xvf /tmp/kvm
mp112# ls
adb                libkvmn.so.0.3patat    sys
arch               libkvm_p a  showrev             trace
boot              m68k             sparc              u370
config            machine          stand              u3b
crash             mc68010          mc68010            u3b15
eeprom            mc68020          sun2               u3b2
format            mdec             sun3               u3b5
fuser             modload          sun386             unixname2bootname
getcons           mps              sun3x              vax
i386              mpstat           sun4               w
iAPX286           pdp11            sun4c
libkvm.a           ps               sun4m
```

Creating client '564_1'

Updating the server's databases

Setting up server file system to support client

Making client's swap file

Setting up client's file system

Copying /export/exec/proto.root.sunos 4.1.4 to /export/root.564_1

Copying binaries

Updating client's databases

Setting up Client's ttytab

7. On the server: Declare /export/exec/kvm/sun4m.sunos.4.1.4
in /etc/exports;
(check line #6,

/usr

/home

/var,spool/mail

/export/root/564_1 -access=564_1,root=564_1

/export/swap.564_1 -access=564_1,root=564_1

/export/exec/kvm/sun4m.sunos.4.1.4 -rw=564_1,access=564_1,root=564_1

~

~

"/etc/exports" 6 lines, 184 characters

8. Update the client's /etc/fstab file:

```
mp112:/export/root/564_1      / nfs rw 0 0
mp112 /export/exec/sun4 sunos 4 1 4      /usr nfs ro 0 0
mp112:/export/exec/kvm/sun4m.sunos.4.1.4      /usr/kvm nfs rw ro 0 0
mp112 ,home mp112            /home/mp112 nfs rw 0 0
mp112:/var/spool/mail        /var/spool/mail nfs rw 0 0
mp112:/export/share/sunos.4.1.4 /usr/share nfs rw 0 0
-
~
"/etc/fstab" 6 lines, 298 characters
```

9. Boot the client.

ok boot net

Resetting ..

zs is initialized

Initializing TLB

Initializing Swift Cache

Allocating SRMMU Context Table

Setting SRMMU Context Register

Setting SRMMU Context Table Pointer Register

Allocating SRMMU Level 1 Table

Level 1 Table allocated, Available Memory 0x03feec00

Mapping RAM

Mapping ROM

ttya initialized

1

SPARCstation 564, No Keyboard

ROM Rev. 2.14.4, 64 MB memory installed, Serial #4096.

Ethernet address 0:80:b6:1:2:3, Host ID: 80001000.

Rebooting with command: net

Boot device: s0 commu sbus ledma04,8400010/1e84 8c000000 File and args

Automatic network cable selection succeeded Using TP Ethernet Interface
1d400

Automatic network cable selection succeeded . Using TP Ethernet Interface

Using IP Address 198.211.242.188 = C6D3F2BC

hostname: 564_1

domainname: norame

server name 'mpll2'

root pathname '/export/root/564_1'

root on mpll2:/export/root/564_1 fstype nfs

Boot: vmunix

Size: 1548288+464288+225760 bytes

VAC ENABLED

SunOS Release 4.1.4 (GENERIC) #1: Fri Jun 16 17:00:30 PDT 1995

Copyright (c) 1983-1993, Sun Microsystems, Inc.

cpu = THEMIS.564

mod0 = FMI MB86904 (md = 0)

mem = 65220K (0x3fb1000)

avail mem = 61083648

entering uniprocessor mode

Ethernet address = 0:80:b6:1:2:3

espdma0 at SBus slot 4 0x8400000

esp0 at SBus slot 4 0x8800000 pri 4 (onboard)

esp0: Warning- no devices found for this SCSI bus

SUNW,bpp0 at SBus slot 4 0xc800000 pri 3 (sbus level 2)

ledma0 at SBus slot 4 0x8400010

le0 at SBus slot 4 0x8c00000 pri 6 (onboard)

Warning: Out of range register specification in device node <flash>

Warning: Out of range register specification in device node <flash>

Warning: Out of range register specification in device node <flash>

zs0 at SBus slot 4 0x1100000 pri 12 (onboard)

zs1 at SBus slot 4 0x1000000 pri 12 (onboard)

SUNW,fdtwo0 at SBus slot 4 0x1400000 pri 11 (onboard)

```

SBVME address 1080000 {vme32d32} not found in VME range table, ignoring
isc
SBVME address 1080400 {vme32d32} not found in VME range table, ignoring
isc
SBVME address 1080800 {vme32d32} not found in VME range table ignoring
isc
SBVME address 1080c00 {vme32d32} not found in VME range table ignoring
isc
SBVME address 1081000 {vme32d32} not found in VME range table, ignoring
isc
SBVME address 1000000 {vme32d32} not found in VME range table, ignoring
mcp
SBVME address 1010000 {vme32d32} not found in VME range table; ignoring
mcp
SBVME address 1020000 {vme32d32} not found in VME range table; ignoring
mcp
SBVME address 1030000 {vme32d32} not found in VME range table; ignoring
mcp
SBVME address 2000000 {vme32d32} not found in VME range table ignoring
mcp
SBVME address 2010000 {vme32d32} not found in VME range table, ignoring
mcp
SBVME address 2020000 {vme32d32} not found in VME range table; ignoring
mcp
SBVME address 2030000 {vme32d32} not found in VME range table ignoring
mcp.
hostname: 564_1
domainname: noname
root on mp112:/export/root/564_1 fstype nfs
swap on mp112:/export/swap/564_1 fstype nfs size 16384K
dump on mp112:/export/swap/564_1 fstype nfs size 16372K
checking filesystems
Automatic reboot in progress...
Wed Nov 15 15:14:05 PST 1995
checking quotas: done.
starting rpc port mapper.
starting RPC key server.
network interface configuration:
le0- flags=63<UP,BROADCAST,NOTRAILERS,RUNNING>
inet 198.211.242.188 netmask ffffffff broadcast 198 211 242.0

```

```

ether 0 80 b6.1.2.3
lo0: flags=49<UP,LOOPBACK,RUNNING> inet 127.0.0.1 netmask ff000000
rdate mp112
Wed Nov 15 15:14 15 1995
mount -vat nfs
mount: mp112:/export/exec/sun4.sunos 4 1.4 already mounted
mount: mp112:/export/exec/kvm/sun4m.sunos.4.1.4 already mounted
mp112 /export/share/sunos.4.1.4 mounted on /usr/share
mp112 home/mp112 mounted on /home/mp112
mp112:/var/spool/mail mounted on /var/spool/mail
starting additional services: biod.
starting system logger
starting local daemons: auditd sendmail statd lockd.
link-editor directory cache
preserving editor files
clearing /tmp
standard daemons: update cron uucp.
starting network daemons: inetd printer
Wed Nov 15 15:14:19 PST 1995

564_1 login:
564_1 login: root
Nov 15 15:14:50 564_1 login: ROOT LOGIN console
Last login: Wed Nov 15 14:44:49 on console
SunOS Release 4.1.4 (GENERIC) #1: Fri Jun 16 17 00:30 PDT 1995
564_1#
564_1#
564_1#
564_1#
564_1# df
Filesystem                kbytes    used    avail capacity  Mounted on
mp112:/export/root/564_1
                        189383    59730   110715     35%      /
mp112:/export/exec/sun4.sunos.4.1.4
                        710110   162944   476155     25%     /usr
mp112:/export/exec/kvm/sun4m.sunos.4.1.4
                        189383    59730   110715     35%   /usr/kvm
mp112:/export/share/sunos.4.1.4
                        710110   162944   476155     25%   /usr/share
mp112 home mp112         710110   162944   476155     25%  /home mp112
mp112 /var/spool/mail

```

```
189383 59730 110715 35% /var/spool/mail
564_1#
10. Download the VME patch on the client, and install them .
11 Reboot the client. If it's a THEMIS 5-64, you need to run "add_vme"
    under OBP, to declare your A32 boards, that will be
    accessed by the 5_64.
```

TN102 VME configuration on SPARC 10MP

Object: Configure VME interface

Boards SPARC 10MP

Operating system SOLARIS 2.3, 2.4

1) Default values for VMEBus request level, size of A32 slave window, and base address of A32 slave window can be overridden by OBP properties.

These properties are respectively named "bus-request-level", "a32-size", and "a32-base". They are replicates of the corresponding bit fields in the VME Master and Slave Configuration registers described at the end of this note.

Generally speaking, the command syntax for setting an OBP property is:

<hex value to set> xdrint " <property-name>" attribute

Notice that the space character between the quote and the property name is mandatory.

Such settings can be made permanent in NVRAM for automatic execution, by use of "nvedit". See manual "OpenBoot Command Reference" for a description of nvedit.

Example: Set Bus request level to 2, A32 slave address to 0x80000000, A32 slave window size to 32 MB.

ok nvedit

1. cd /THEMIS,vme

2. 2 xdrint " bus-request-level" attribute

3. 80 xdrint " a32-base" attribute

```

4. 1 xdrint " a32-size" attribute
5 ^C
   ok nvstore
   ok setenv use-nvramcc? true
   ok reset

```

(**<Caution>** the last command will reset the board !)

Changes may be examined by

```

ok cd /THEMIS,vme
ok .attributes

```

Note that A32 slave access is normally left disabled. Use patch_kmem below to enable it

2) Other default system values for configuration of VME interface can be overridden by patching MVIC registers from applications running in user space

As a matter of fact MVIC registers are accessible in /dev/kmem, starting at location 0xFFED B000

Thus address corresponds to the "address" property, as defined by OBP in the "/THEMIS,vme" device node.

A sample C file for patching /dev/kmem, "patch_kmem.c" is included in this note. It must be run in superuser mode

```

patch_kmem ffedbf0
-> Read VMEbus master configuration register

```

```

patch_kmem ffedbf8 2800000
-> Set VME A32 slave address to 0x8000.0000,
    VME A32 slave window size to 64 MB

```

VMEbus Master Configuration Register (offset 0xFF0)

=====

bit [31]VME Slave A32 Address space enable

If set to 1, slave access in A32 address space is enabled

bit [30] VME Slave A24 Address space enable

If set to 1, slave access in A24 address space is enabled

bit [29] VME Slave A16 Address space enable

If set to 1, slave access in A16 address space is enabled

Always to be left disabled under Solaris.

bits [28:15] Reserved

bits [14:11] A24 Slave Base Address

These 4 bits determine where in the A24 slave address space the 1MB DVMA region will respond to. Note that DVMA (Direct Virtual Memory Access) is reserved for SunOS device drivers.

Bits [14:11] correspond to bits [23:20] of VME A24 address

bits [10:8] Reserved

bits [7:6] VMEbus DTACK time-out

This field determines how long the MVIC waits for a slave to respond with a DTACK before asserting a Bus Error on the VMEbus

bits [7:6] time-out

0 0 51.2 us

0 1 12.8 us

1 0 3.2 us

1 1 none

bits [5:4] VMEbus Bus Request time-out

This field determines how long the MVIC waits to acquire the VMEbus before asserting a Bus Error on the VMEbus or halting the DMA transfer

bits [5:4] time-out

0 01638.4 us
 0 1 409.6 us
 1 0 12.8 us
 1 1 none

bits [3:2] VMEBus Request Level

bits [3:2] Bus Request Level

 0 0 0
 0 1 1
 1 0 2
 1 1 3

bit [1] VMEbus Arbiter mode

When set to 1, the VMEbus will arbitrate the bus requests in a round-robin manner. Otherwise a fixed priority scheme will be used

bit [0] Reserved

VMEbus Slave Configuration Register (offset 0xFE8)

=====

bits [31:28] Reserved

bits [27:24] A32 VME Slave Window Size

bits [27:24] A32 Slave Window Size

 0 16 MB
 1 32 MB
 2 128 MB
 3 256 MB
 4 512 MB

The board will respond if the A32 enable bit is set and
 Window Start Address + Size > VME Addr >= Window Start Address

bits [23:16] A32 Slave Base Address

Determines the starting VMEbus address the board will respond to in A32 slave mode. These 8 bits correspond to bits [31:24] of VME A32 address.

bits [15:0]Reserved

(Note that all "Reserved" bits shall not be modified)

```

----- Cut here for patch_kmem.c -----
/*
 * PW 11/93. User access to kmem
 *

#include <fcntl.h>
#include <errno.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/mman.h>

#define DATATYPE unsigned int

void main(argc,argv)
int argc;
char *argv[];
{
    int kmem;
    caddr_t pageptr;
    DATATYPE *ptr, oldvalue;
    unsigned addr, value;

/* Check args */
    if (argc < 2) {
usage
        printf("%s <hex address> [<hex value>]\n",argv[0]);
        exit(0);
    }
    sscanf(argv[1],"%x",&addr);

```

```

        if (argc == 3)                /* value to set */
            sscanf(argv[2], "%x", &value);

/*
 * open kmem
 */
    if ((kmem = open("/dev/kmem", O_RDWR)) < 0) {
        perror("open error");
        exit(errno);
    }

    *
    * map it, at a page boundary
    */
    if ((pageptr = mmap(0L, PAGE_SIZE, PROT_READ|PROT_WRITE,
                        MAP_SHARED, kmem, (off_t)(addr & PAGEMASK)), ==
(caddr_t)-1) {
        perror("mmap error");
        exit(errno);
    }
    ptr = (DATATYPE *)(((unsigned int)pageptr + (addr & PAGEOFFSET

/*
 * Operations on kmem
 */
    oldvalue = *ptr;
    if (argc == 3)                /* value to set */
        printf("0x%x : 0x%x --> 0x%x\n", addr, oldvalue, "ptr= DATATYPE value :
    else
        printf("0x%x : 0x%x\n", addr, oldvalue);

/*
 * unmap kmem
 */
    if (munmap(pageptr, PAGE_SIZE) == -1) {
        perror("munmap error");
        exit(errno);
    }
}

----- Cut here --

```

7.7 Removing NEXUS Driver

TECHNICAL NOTE No 131-00
SUBJECT: Removing NEXUS driver on the 10MP
BOARD: THEMIS 10MP
O.S.: Solaris 2.4
Hardware rev: all
DATE: Oct 2, 1995

Since release 1.7, THEMIS solaris 2.4vme NEXUS driver is released in the pkgadd format. Some problems may occur if you are trying to run "pkgadd -d /dev/rmt/0" (package installation) on a system to which the VME NEXUS driver was installed using the former tvme-install install script.

The workaround is to follow those instructions:

1. Boot single user
2. "rem_drv tvme" -- ignore any error messages that might result, such as "device busy"
3. "rem_drv mvcc" -- again, ignore any error messages
4. Restore /etc/system from /etc/system.bak. /etc/system.bak was created by the tvme-install script. If other changes have been made to /etc/system since tvme-install was run, then the lines related to the tvme driver should be removed manually, since restore from /etc/system.bak will result in loss of those changes. The relevant lines are:

```
moddir /kernel /usr/kernel /themis
forceload drv/tvme
```

5. Restore /etc/driver_classes or /kernel/drv/classes from the backup file, which will name /etc/driver_classes.bak or /kernel/drv/classes.bak. Again, if those files have been modified since tvme-install was run, the lines related to the tvme driver should be removed manually. The relevant line is:

```
"THEMIS,vme"vme
```

6. Restore /etc/devlink.tab from /etc/devlink.tab.bak. Once again, be sure the file hasn't been modified since tvme-install was run, or else remove the lines related to tvme manually. The relevant lines are:

```
type=ddi_pseudo,name=mvc;minor=amvc0
type=ddi_pseudo,name=vmemem;minor=a32d32    vme32d32
type=ddi_pseudo,name=vmemem;minor=a24d32    vme24d32
type=ddi_pseudo,name=vmemem;minor=a16d32    vme16d32
type=ddi_pseudo,name=vmemem;minor=a32d16    vme32d16
type=ddi_pseudo,name=vmemem;minor=a24d16    vme24d16
type=ddi_pseudo,name=vmemem;minor=a16d16    vme16d16
```

7. Reboot the system using the '-r' flag.
8. Run the "pkgadd -d /dev/rmt.0" command to install the new package. Then reboot the system using the '-r' flag.
9. Run the "pkginfo -l" to make sure the package was installed correctly.

7.8 Determining Speed of MBus Modules

TECHNICAL NOTE No 132-00
SUBJECT: Determining Speed of Ross MBUS modules.
BOARD: THEMIS 10MP
O.S.: all
Hardware rev: all
DATE: Oct 5, 1995

The proper way of determining the Frequency of an MBUS CPU module is to use the module-info command under OBP. Unfortunately, in the 10MP module-info is printing a wrong value of the CPL frequency. The value is displayed using 2 digits only. So the upper digits will not be displayed. Say, you have a 100MHz module. Then module-info will return "00 Hnz".

There is no special registers on the Ross modules, that tell users what is the frequency of the module they are using. OBP is timing a software loop to determine it.

The only way to find the frequency is to check the label on the Ross module MBUS connector. The first number of the left is giving the value (66, 90, 100, ...).

7.9 Memory Error Message

TECHNICAL NOTE No 134-00
SUBJECT: "le0: memory error" warning message.
BOARD: THEMIS 10MP
O.S.: Solaris 2.4
Hardware rev: all
DATE: August 1, 1995

Under Solaris 2.4, and when the 10MP is making intensive SBUS and VME activity, the message:

le0: memory error

will pop-up at random occasions. This is due to the Lance (Ethernet) driver that is unable to access memory because of the system activity.

This problem is solved by a Sun patch, that deals with the latency time the driver can wait before getting access to memory. See below for a readme of that patch.

To install that patch

```
# cd 101979-06  
# ./installpatch
```

To check if this patch is already installed

```
# showrev -p
```

To get that patch, or in case of problems.

THEMIS Technical Support
3185, Laurelview Court Fremont, CA 94538 USA
(510)252-0870
(510)490-5529
support@themis.com

Patch-ID# 101979-06

Keywords: ethernet driver memory leak ethernet auto hang

Synopsis: SunOS 5.4: le driver fixes

Date: Aug/22/95

Solaris Release: 2.4

SunOS Release: 5.4

Unbundled Product

Unbundled Release

Topic: SunOS 5.4: le driver fixes

BugId's fixed with this patch: 1145294 1161058 1171562 1177296

Changes incorporated in this version: 1177296

Relevant Architectures: sparc

Patches accumulated and obsoleted by this patch: 102444-01

Patches which conflict with this patch

Patches required with this patch.

Obsoleted by

Files included with this patch:

/kernel/drv/le

Problem Description:

1177296 ethernet interface hangs on large transfers over AUI but not TP

This is a re crank of 101979-05. -05 rev was missing 1177296 fix

(from 101979-05)

1177296 ethernet interface hangs on large transfers over AUI but not TP

(from 101979-04)

1161058 Under many collisions, get erroneous le0: No carrier message

Under many collisions, get erroneous le0: No carrier message

(from 101979-03)

1171562 Getting "le0: Memory error" when copying large file across network

SS10 with dual processors running 2.3 with OLDS 2.0.1 installed, gets le0: Memory error when copying large files over the network. A hard hang then occurs. This eliminates console error message that existed with 101979-02

(from 101979-02)

1171562 Getting "le0: Memory error" when copying large file across network

SS10 with dual processors running 2.3 with OLDS 2.0.1 installed, gets le0: Memory error when copying large files over the network. A hard hang then occurs.

(from 101979-01)

**I145294 classic 4.1 3C freeze 5-10Mins on receipt of Giant Packet from
ff ff ff ff ff ff**

Problem is seen on active networks where the network interface on Classic's, LX's, Sparcstations 5 and 20 (based on MACIO) may hang and ignore incoming ethernet packets after receipt of a giant packet that is greater than 4096 bytes. This fix will detect this condition and reset the ethernet interface to prevent the interface from hanging.

Patch Installation Instructions.

Generic 'installpatch' and 'backoutpatch' scripts are provided within each patch package with instructions appended to this section. Other specific or unique installation instructions may also be necessary and should be described below.

Special Install Instructions.

None

Instructions to install patch using "installpatch"

1. Become super-user
2. Apply the patch by typing

```
<dir>installpatch <patch-dir>
```

where <dir> is the directory containing installpatch, and <patch-dir> is the directory containing the patch itself.

Example

```
# cd /tmp/123456-01
# ./installpatch
```

3. If any errors are reported, see "Patch Installation Errors" in the Command Descriptions section below.

Instructions for installing a patch on a dataless client

- 1 Before applying the patch, the following command must be executed on the server to give the client read-only, root access to the exported /usr file system so that the client can execute the pkgadd command

```
share -F nfs -o ro,anon=0 /export/exec/<os_version>/usr
```

The command

```
share -F nfs -o ro,root=<client_name> \  
/export/exec/<os_version>/usr
```

accomplishes the same goal, but only gives root access to the client specified in the command

- 2 Login to the client system and become super-user
- 3 Continue with step 2 in the "Instructions to install patch using installpatch" section above.

Instructions for installing a patch on a diskless client

****** To install a patch on a diskless client, you may either follow the instructions for installing on a dataless client (that is, you may logon to the client and install the patch), or you may use the following instructions to install the patch while on the server

- 1 Find the complete patch for the root directory of the diskless client
- 2 Install the patch normally, but add the command option **R<path>** to the command line. **<path>** should be completely specified

Example:

```
# cd /tmp/123456-01  
# /installpatch -R /export/root/client1
```

Instructions for backing out a patch using "backoutpatch"

1. Become super-user
2. Change directory to /var/sadm/patch.
`cd /var/sadm/patch`
3. Backout patch by typing
`<patch-id>/backoutpatch <patch-id>`
where <patch-id> is the patch number

Example

```
# cd /var/sadm/patch
# 123456-01/backoutpatch 123456-01
```

4. If any errors are reported, see "Patch Backout Errors" in the Command Description section below

Instructions for backing out a patch on a dataless client

1. Give the client root access to /usr as specified in the installpatch section
2. Logon to the client and follow backoutpatch instructions as specified above

Instructions for backing out a patch on a diskless client

** To backout a patch on a diskless client, you may either follow the instructions for backout on a dataless client (that is, you may logon to the client and backout the patch), or you may use the following instructions to backout the patch while on the server

1. Find the complete path for the root directory of the diskless client
2. Backout the patch normally, but add the command option `-R <path>` to the command line. `<path>` should be completely specified

Example

```
# cd /export/root/client1/var/sadm/patch
# /123456-01/backoutpatch -R /export/root/client1 123456-01
```

Instructions for identifying patches installed on system

Patch packets that have been installed can be identified by using the `-p` option. To find out which patches are installed on a diskless client, use both the `-R <dir>` option and the `-p` option, where `<dir>` is the fully specified path to the client's root directory

```
#cd /tmp/123456-01
# /installpatch -p
# /installpatch -R /export/root/client1 -p
```

Also note that the command `'showrev -p'` will show the patches installed on the local machine but will not show patches installed on clients

Command Descriptions

NAME

installpatch - apply patch package to Solaris 2.x system
backoutpatch - remove patch package, restore previously saved files

SYNOPSIS

installpatch [-udpV] [-S <service>] <patch number>
backoutpatch [-fv] [-S <service>] <patch number>

DESCRIPTION

These installation and backout utilities apply only to Solaris 2.x associated patches. They do not apply to Solaris 1.x associated patches. These utilities are currently only provided with each patch package and are not included with the standard Solaris 2.x release software

OPTIONS

installpatch.

- u** unconditional install, turns off file validation. Allows the patch to be applied even if some of the files to be patched have been modified since original installation
- d** Don't back up the files to be patched. This means that the patch **CANNOT BE BACKED OUT**
- p** Print a list of the patches currently applied
- V** Print script version number
- S** <service> Specify an alternate service (e.g. Solaris_2.3) for patch package processing references
- R** <dir> Specify an alternate package installation root. Most useful for installing patches on diskless clients while logged on to the server

backoutpatch:

- f** force the backout regardless of whether the patch was superseded
- V** print version number only
- S** <service> Specify an alternate service (e.g. Solaris_2.3) for patch package processing references
- R** <dir> Specify an alternate package installation root. Most useful for removing patches on diskless clients while logged on to the server

DIAGNOSTICS**Patch Installation Errors.**
-----**Error message:**

The prepatch script exited with return code <retcode>
Installpatch is terminating.

Explanation and recommended action: The prepatch script supplied with the patch exited with a return code other than 0. Run a script trace of the installpatch and find out why the prepatch had a bad return code. Fix the problem and re-run installpatch

To execute a script trace:

```
# sh -x ./installpatch . > /tmp/patchout 2>&1
```

The file /tmp/patchout will list all commands executed by installpatch. You should be able to determine why your prepatch script failed by looking through the /tmp/patchout file. If you still can't determine the reason for failure, contact customer service.

Error message:

The postpatch script exited with return code <retcode>
Backing out patch.

Explanation and recommended action: The postpatch script provided with the patch exited with an error code other than 0, and the patch has not previously been applied. Installpatch will execute backoutpatch to return the system to its pre-patched state. Create a script trace of the installpatch (see above) and find out why the postpatch script failed. Correct and re-execute installpatch. If you are unable to determine why the postpatch script failed, contact customer service.

Error message:

The postpatch script exited with return code <retcode>
 Not backing out patch because this is a re-installation.
 The system may be in an unstable state!
 Installpatch is terminating

Explanation and recommended action: The postpatch script provided with the patch exited with an error code other than 0. Because this is a re-installation of a patch, installpatch will not automatically backout the patch. You may backout the patch manually using the backoutpatch command, then generate a script trace of the installpatch as described above. Find out why the postpatch failed, correct the problem, and re-install the patch. If you are unable to determine why the postpatch script failed, contact customer service.

Error message:

Patch <patch-id> has already been applied

Explanation and recommended action: This patch has already been applied to the system and no additional patch packages would be added due to a re-installation. If the patch has to be reapplied for some reason, backout the patch and then reapply it.

Error message:

Symbolic link in package <pkg>
 Symbolic links can't be part of a patch
 Installpatch is terminating

Explanation and recommended action: The patch was incorrectly built. Contact customer service to get a new patch

Error message:

This patch is obsoleted by patch <patch-id> which has already been applied to this system. Patch installation is aborted

Explanation and recommended action: Occasionally, a patch is replaced by a new patch which incorporates the bug fixes in the old patch and supplies additional fixes also. At this time, the earlier patch is no longer made available to users. The second patch is said to "obsolete" the first patch. However, it is possible that some users may still have the earlier patch and try to apply it to a system on which the later patch is already applied. If the obsoleted patch were allowed to be applied, the additional fixes supplied by the later patch would no longer be available, and the system would be left in an inconsistent state. This error message indicates that the user attempted to install an obsoleted patch. There is no need to apply this patch because the later patch has already supplied the fix

Error Message:

None of the packages to patch are installed on this system

Explanation and recommended action: The original packages for this patch have not been installed and therefore the patch cannot be applied. The original packages need to be installed before applying the patch.

Error message:

This patch is not applicable to client systems

Explanation and recommended action: The patch is only applicable to servers and standalone machines. Attempting to apply this patch to a client system will have no effect on the system

Error message

The **-S** and **-R** arguments are mutually exclusive

Explanation and recommended action: You have specified both a non-native service to patch, and a package installation root. These two arguments are mutually exclusive. If patching a non-native `usr` partition, the **-S** option should be used to patch all clients using that service. If patching a client's root partition (either native or non-native), the **-R** option should be used.

Error message

The `<service>` service cannot be found on this system.

Explanation and recommended action: You have specified a non-native service to patch, but the specified service is not installed on your system. Correctly specify the service when applying the patch.

Error message:

The Package Install Root directory `<dir>` cannot be found on this system.

Explanation and recommended action: You have specified a directory that is either not mounted, or does not exist on your system. Specify the directory correctly when applying the patch.

Error message:

The `/usr/sbin/pkgadd` command is not executable.

Explanation and recommended action: The `/usr/sbin/pkgadd` command cannot be executed. The most likely cause of this is that `installpatch` is being run on a diskless or dataless client and the `/usr` file system was not exported with root access to the client. See the section above on "Instructions for installing a patch on a diskless or dataless client."

Error message

<patch-id> packages are not proper patch packages

Explanation and recommended action: The patch directory supplied as an argument to installpatch did not contain the expected package format. Verify that the argument supplied to installpatch is correct.

Error message:

The following validation error was found

<validation error(s)>

Explanation and recommended action: Before applying the patch, the patch application script verifies that the current versions of the files to be patched have the expected fcs checksums and attributes. If a file to be patched has been modified by the user, the user is notified of this fact. The user then has the opportunity to save the file and make a similar change to the patched version. For example, if the user has modified /etc/inet/inetd.conf and /etc/inet/inetd.conf is to be replaced by the patch, the user can save the locally modified /etc/inet/inetd.conf file and make the same modification to the new file after the patch is applied. After the user has noted all validation errors and taken the appropriate action for each one, the user should re-run installpatch using the "-u" (for "unconditional") option. This time, the patch installation will ignore validation errors and install the patch anyway.

Error message:

Insufficient space in /var/sadm/patch to save old files

Explanation and recommended action: There is insufficient space in the /var/sadm/patch directory to save old files. The user has two options for handling this problem: (1) generate additional disk space by deleting unneeded files, or (2) override the saving of the old files by using the "-d" (do not save) option when running installpatch. However, if the user elects not to save the old versions of

the files to be patched, backoutpatch CANNOT be used

One way to regain space on a system is to remove the save area for previously applied patches. Once the user has decided that it is unlikely that a patch will be backed out, the user can remove the files that were saved by installpatch. The following commands should be executed to remove the saved files for patch xxxxxx-yy

```
cd /var/sadm/patch/xxxxxx-yy
rm -r save/*
rm oldhlessaved
```

After these commands have been executed, patch xxxxxx-yy can no longer be backed out.

Error message

Save of old files failed

Explanation and recommended action: Before applying the patch, the patch installation script uses cpio to save the old versions of the files to be patched. This error message means that the cpio failed. The output of the cpio would have been preceded this message. The user should take the appropriate action to correct the cpio failure. A common reason for failure will be insufficient disk space to save the old versions of the files. The user has two options for handling insufficient disk space (1) generate additional disk space by deleting unneeded files, or (2) override the saving of the old files by using the "-d" option when running installpatch. However if the user elects not to save the old versions of the files to be patched, the patch CANNOT be backed out

Error message

Pkgadd of <pkgname> package failed with error code <code>
See /tmp/log.<patch-id> for reason for failure

Explanation and recommended action: The installation of one of patch packages failed. Installpatch will backout the patch

to leave the system in its pre-patched state. See the log file for the reason for failure. Correct the problem and re-apply the patch.

Error message

Pkgadd of <pkgname> package failed with error code <code>
Will not backout patch...patch re-installation
Warning: The system may be in an unstable state!
See /tmp/log.<patch-id> for reason for failure

Explanation and recommended action The installation of one of the patch packages failed. Installpatch will NOT backout the patch. You may manually backout the patch using backoutpatch, then re-apply the entire patch. Look in the log file for the reason pkgadd failed. Correct the problem and re-apply the patch.

Patch Installation Messages

Note: the messages listed below are not necessarily considered errors as indicated in the explanations given. These messages are, however, recorded in the patch installation log for diagnostic reference.

Message:

Package not patched.
PKG=SLNxxxx
Original package not installed

Explanation One of the components of the patch would have patched a package that is not installed on your system. This is not necessarily an error. A Patch may fix a related bug for several packages. Example: suppose a patch fixes a bug in both the online-backup and fddi packages. If you had online-backup installed but didn't have fddi installed, you would get the message

Package not patched.
PKG=SUNWbf
Original package not installed

This message only indicates an error if you thought the package was installed on your system. If this is the case, take the necessary action to install the package, backout the patch (if it installed other packages) and re-install the patch.

Message

```
Package not patched
PKG=SLNxxx
ARCH=xxxxxxx
VERSION=xxxxxxx
Architecture mismatch
```

Explanation One of the components of the patch would have patched a package for an architecture different from your system. This is not necessarily an error. Any patch to one of the architecture specific packages may contain one element for each of the possible architectures. For example, Assume you are running on a sun4m. If you were to install a patch to package SUNWcar, you would see the following (or similar) messages:

```
Package not patched.
PKG=SUNWcar
ARCH=sparc_sun4c
VERSION=11.5.0,REV=2.0.18
Architecture mismatch
```

```
Package not patched
PKG=SUNWcar
ARCH=sparc_sun4d
VERSION=11.5.0,REV=2.0.18
Architecture mismatch
```

```
Package not patched.
PKG=SUNWcar
ARCH=sparc_sun4e
VERSION=11.5.0,REV=2.0.18
Architecture mismatch
```

```
Package not patched
PKG=SUNWcar
```

```
ARCH=sparc.sun4
VERSION=11.5.0,REV.2.0.18
Architecture mismatch
```

The only time these messages indicate an error condition is if installpatch does not correctly recognize your architecture

Message:

```
Package not patched
PKG=SUNxxxx
ARCH=xxxx
VERSION=xxxxxxx
Version mismatch
```

Explanation: The version of software to which the patch is applied is not installed on your system. For example, if you were running Solaris 5.3 and you tried to install a patch against Solaris 5.2, you would see the following (or similar) message:

```
Package not patched
PKG=SUNWcsu
ARCH=sparc
VERSION=10.0.2
Version mismatch
```

This message does not necessarily indicate an error. If the version mismatch was for a package you needed patched, either get the correct patch version or install the correct package version. Then backout the patch (if necessary) and re-apply.

Message:

```
Re-installing Patch.
```

Explanation: The patch has already been applied, but there is at least one package in the patch that could be added. For example, if you applied a patch that had both Openwindows and Answerbook components, but your system did not have Answerbook installed, the Answerbook parts of the patch would not have been applied. If, at a later time, you pkgadd Answerbook, you could re-apply the patch, and the Answerbook components of the

patch would be applied to the system

Message

Installpatch Interrupted.
Installpatch is terminating

Explanation: Installpatch was interrupted during execution (usually through pressing ^C) Installpatch will clean up its working files and exit

Message:

Installpatch Interrupted.
Backing out Patch

Explanation: Installpatch was interrupted during execution (usually through pressing ^C) Installpatch will clean up its working files, backout the patch, and exit

Patch Backout Errors.

Error message

prebackout patch exited with return code <retcode>
Backoutpatch exiting

Explanation and corrective action: the prebackout script supplied with the patch exited with a return code other than 0. Generate a script trace of backoutpatch to determine why the prebackout script failed. Correct the reason for failure, and re-execute backoutpatch.

Error message

postbackout patch exited with return code <retcode>
Backoutpatch exiting "

Explanation and corrective action: the postbackout script supplied with the patch exited with a return code other than 0. Look at the postbackout script to determine why it failed. Correct the failure and, if necessary, RE-EXECUTE THE POSTBACKOUT SCRIPT ONLY

Error message

Only one service may be defined.

Explanation and corrective action: You have attempted to specify more than one service from which to backout a patch. Different services must have their patches backed out with different invocations of backoutpatch.

Error message

The -S and -R arguments are mutually exclusive.

Explanation and recommended action: You have specified both a non-native service to backout, and a package installation root. These two arguments are mutually exclusive. If backing out a patch from a non-native usr partition, the -S option should be used. If backing out a patch from a client's root partition (either native or non-native), the -R option should be used.

Error message

The <service> service cannot be found on this system.

Explanation and recommended action: You have specified a non-native service from which to backout a patch, but the specified service is not installed on your system. Correctly specify the service when backing out the patch.

Error message:

Only one rootdir may be defined.

Explanation and recommended action: You have specified more than one package install root using the -R option. The -R option may be used only once per invocation of backoutpatch.

Error message

The <dir> directory cannot be found on this system.

Explanation and recommended action: You have specified a directory using the **-R** option which is either not mounted, or does not exist on your system. Verify the directory name and re-backout the patch.

Error message

Patch <patch-id> has not been successfully applied to this system

Explanation and recommended action: You have attempted to backout a patch that is not applied to this system. If you must restore previous versions of patched files, you may have to restore the original files from the initial installation CD

Error message

Patch <patch-id> has not been successfully applied to this system
Will remove directory <dir>

Explanation and recommended action: You have attempted to back out a patch that is not applied to this system. While the patch has not been applied, a residual `/var/sadm/patch/<patch-id>` (perhaps from an unsuccessful `installpatch`) directory still exists. The patch cannot be backed out. If you must restore old versions of the patched files, you may have to restore them from the initial installation CD

Error message:

This patch was obsoleted by patch <patch number>
Patches must be backed out in the order in
which they were installed. Patch backout aborted

Explanation and recommended action: You are attempting to backout patches out of order. Patches should never be backed-out out of sequence. This could undermine the integrity of the more current patch.

Error message

Patch <patch-id> was installed without backing up the original files. It cannot be backed out

Explanation and recommended action. Either the -d option of installpatch was set when the patch was applied, or the save area of the patch was deleted to regain space. As a result, the original files are not saved and backoutpatch cannot be used. The original files can only be recovered from the original installation CD.

Error message
pkgrm of <pkgname> package failed return code <code>
See /var/sadm/patch/<patch-id>/log for reason for failure

Explanation and recommended action. The removal of one of patch packages failed. See the log file for the reason for failure. Correct the problem and run the backout script again.

Error message
Restore of old files failed

Explanation and recommended action. The backout script uses the cpio command to restore the previous versions of the files that were patched. The output of the cpio command should have preceded this message. The user should take the appropriate action to correct the cpio failure.

KNOWN PROBLEMS.

On client server machines the patch package is NOT applied to existing clients or to the client root template space. Therefore, when appropriate, ALL CLIENT MACHINES WILL NEED THE PATCH APPLIED DIRECTLY USING THIS SAME INSTALLPATCH METHOD ON THE CLIENT. See instructions above for applying patches to a client.

A bug affecting a package utility (eg pkgadd pkgrm pkgchk) could affect the reliability of installpatch or backoutpatch which uses package utilities to install and backout the patch package. It is recommended that any patch that fixes package utility problems be reviewed and, if necessary, applied before other patches are applied. Such existing patches are:

100901Solaris 2.1

101122Solaris 2.2

101331Solaris 2.3

SEE ALSO

pkgadd, pkgchk, pkgm, pkginfo, showrev, cpio

7.10 VME Bus Interrupt Problem

TECHNICAL NOTE No 135-00
SUBJECT: VME Bus error and interrupt problem on 10MP/solaris2.4
BOARD: THEMIS 10MP
O.S.: Solaris 2.4
Hardware rev: all
DATE: October 1, 1995

There is a problem in the way Solaris 2.4 is handling VME bus errors for HyperSparc MBUS modules. VME bus errors could result in a system crash, when made from a user program. The normal behavior is to send a signal to the task, not to crash the system.

Themis has released a new VME nexus driver (version 1.7) which is fixing that problem.

Also, the nexus driver was modified so as not to LACK VME interrupts which are masked on the 10MP. The problem was to be seen in the following situation.

There is an SBUS interrupt level 3. The 10MP needs to determine if the interrupt is a VME or SBUS interrupt. To do so, it is making an LACK cycle at level 3 on the VME. If the LACK cycle times-out, then the 10MP considers this was an SBUS interrupt. But suppose, that in the same time there was a local SBUS interrupt level 3 on the 10MP, there was also a VME interrupt level 3, from and to 2 other boards. Then, before version 1.7 of the VME nexus driver, the 10MP was going to LACK that interrupt. This was causing some kind of problems as the 10MP was LACKing an interrupt not meant to it. Also the 10MP would print a message, saying it got a Spurious interrupt.

The conclusion is that, it is wise to mask all VME interrupts not

used by the 10MP, as this will suppress the VME IACK cycle during SBUS interrupt. This will also increase interrupt latency for SBUS interrupts.

To check the version of the nexus driver

```
# pkginfo -d THEMISvme
```

To mask interruptopts on the 10MP, please see TN107

7.11 10MP OBP PROM

TECHNICAL NOTE No 138-00
SUBJECT: New 10MP OBP prom (Watchdog reset/Solaris 1.1.2 boot problem)
BOARD: THEMIS 10MP
O.S.: Solaris 1.1.2 and 2.4
Hardware rev: all
DATE: October 18, 1995

The 10MP OBP prom 2.12.4 is solving 2 different problems on the 10MP

1. Under Solaris 1.1.2 - Early ROSS modules (Freq<100MHZ) will not be able to boot solaris 1.1.2.
2. Under Solaris 2.4 - If users "break into" OBP using the L1(STOP)-A keyboard keys, or the VT100 break key, then they won't be able to return to OBP by using the OBP "go" command. Instead they will get a "no program active" error message. Also, after the machine is halted, the message "Watchdog reset" will be printed.

7.12 New 10MP OBP PROM Reset

TECHNICAL NOTE No 139-00
SUBJECT New 10MP OBP prom (Watchdog reset/Solaris 112 boot problem,
BOARD: THEMIS 10MP
O.S. Solaris 1.1.2 and 2.4
Hardware revision
DATE October 18, 1995

The 10MP OBP prom 2.12.4 is solving 2 different problems on the 10MP

1. Under Solaris 1.1.2 - Early ROSS modules (Freq<100MHZ) will not be able to boot solaris 1.1.2.
2. Under Solaris 2.4 - If users "break into" OBP using the L1(STOP)-A keyboard keys or the VT100 break key, then they won't be able to return to OBP by using the OBP "go" command. Instead they will get a "no program active" error message. Also, after the machine is halted, the message "Watchdog reset" will be printed.

7.13 Using TTYC OBP PROM

TECHNICAL NOTE No 145-00
SUBJECT: Using the TTYC OBP prom
BOARD: THEMIS 10MP
O.S.: Solaris 2.x
Hardware rev: all
DATE: Nov, 1995

THEMIS provides a special OBP prom on the 10MP, for customers who want to be able to use ttyc as a console.

After you set the ttyc and ttyd jumpers accordingly (from Keyboard/Mouse to TTYC and TTYD, see 10MP user's manual,

J2501 from 1-2 to 2-3
J2502 from 1-2 to 2-3)

you need to consider the following:

Solaris 2.X will boot under the new OBP without any OS modifications. The system will need to reconfigure devices, so after the new prom is installed, the system should be booted using 'boot -r'. However, Solaris will not push the correct streams drivers onto ttyc by default. The result will be that the terminal modes will not echo correctly or perform canonical input and output processing until the system is in multi-user mode and at a login prompt. Once reaching a login prompt, login as root and edit the file /etc/uucp. Changes the lines referencing zs to the following

```
zs 0 3 ldterm ttcompat
zs 131072 131075 ldterm ttcompat
```

This will cause the system to autopush the appropriate streams modules on ttyc and ttyd.

The software interface provided for the TOMP platform fully supports standard VMEbus device drivers written for Solaris environments. It is the intention of Themis Computer to fully support users who wish to write their own VME device drivers that would function on Themis SPARC TOMP platforms. To aid these users and to illustrate the specific features of the VMEbus architecture, Themis provides a number of sample device drivers. Themis provides the complete source code and the necessary configuration files for these drivers.

- `vmeintr` is a sample driver provided by Themis Computer to illustrate the vectored interrupt mechanism of the VMEbus. The `vmeintr` driver relies on the driver configuration files to specify the interrupts it should handle. Each instance of the driver registers the interrupts with the system. Through the `ioctl`s implemented by the driver, the user can generate interrupts and send them to the appropriate driver instances. The interrupt generator and the interrupt receiver need to run on different computers.
- `vmedvma` is a sample driver provided by Themis Computer to illustrate the use of Direct Virtual Memory Access within a device driver. The user can ask the driver to allocate a DMA region on the VMEbus. The driver allocates all the necessary resources to support a DVMA transfer to this region. The driver also implements mechanisms by which the user program can write to or read from the DVMA region.

- `simpl-edma` is a sample driver provided by Themis Computer to illustrate the use of Direct Memory Access within a device driver. It is a simplified version of the standard `vmedma` driver. This driver is mainly intended for programmers writing device drivers that perform DMA operations on the VMEbus.

The source and binary versions of these drivers can be found in the directory `/opt/THEMISvme/drv`. System programmers that are writing device drivers for VME bus devices are strongly encouraged to go through the source code of these sample drivers. Along with the information found in Chapter 8, Writing Device Drivers, these sample drivers offer a good insight into the design and development of device drivers for VMEbus devices. The drivers can be modified without any restrictions.

The sample device drivers are not essential for the normal operation of the SPARC JOMP system. When the THEMISvme package is installed, the drivers and their configuration files are copied to the disk, but are not installed on the system. After rebooting the machine, the user may choose to install these drivers.

Scripts have been provided to perform the above tasks. The `install` sample script copies the drivers (as they are packaged) to the `/kernel/drv` directory. It then installs the drivers on the system, creates the files under `/dev` and copies the header files into the `/usr/include/themis` directory. The `remove` sample script undoes all the installation done by the `install` script. You are strongly urged to carefully review the scripts before executing them.

If you need to make any changes to the drivers, you need to compile the drivers after the changes. After that you can copy the newly built driver object files to the `/kernel/drv` directory. You can install the new drivers on the running system by first executing the `rem_drv` command and then the `add_drv` command.

The `remove` sample script removes the sample drivers from the running system. The script deconfigures the drivers from the kernel and removes the driver object files from the `/kernel/drv` directory.

Manual Pages

A

This section contains the manual pages for all the drivers provided by Themis Computer. Manual pages are available for the standard nexus and DMA drivers as well as the sample drivers. Online copies of the manual pages for the standard drivers can be found in the standard manual pages location. Online copies of manual pages for the sample drivers can be found in the `/opt/THEMISvme/man` directory.

A

NAME

"THEMIS,vme",tvme - Nexus driver for VME bus

SYNOPSIS

THEMIS,vme3f,ff5ff00c

DESCRIPTION

The tvme device driver is a VMEbus Nexus Driver designed especially for Themis Computer's SPARC10MP platforms. The driver is configured to be the bus nexus driver for the class "vme". All leaf drivers that specify their parent as "vme" will be attached to the tvme driver. The tvme driver provides support for all VMEbus operations, including VME interrupt processing, Direct Virtual Memory Access (DVMA) on the VMEbus, mapping of registers on the VMEbus to kernel space etc.

The tvme driver provides these features:

- Master access in VMEB34 mode
- Slave access in VMEB32 mode
- Direct Virtual Memory Access (DVMA) on the VMEbus
- Block Transfer in Master and Slave modes

The interface between the VMEbus and the MBus is provided by the Fujitsu MB86986 MBus to VMEbus Interface Controller (MVIC). The tvme driver supports a single instance of MVIC. The driver initialises the controller for operation and handles asynchronous faults from the controller. User access to features of the MVIC, including the Direct Memory Access engine is provided by the mvc(7) driver.

The tvme driver supports the vectored interrupt mechanism of the VMEbus architecture. All vme devices that generate interrupts are expected to provide the 'interrupts' property in the device configuration file. The tvme driver supports SPARC interrupt priorities 2,3,5,7,9,11 and 13. The driver handles interrupt vectors 0 to 255.

The driver fully supports Direct Virtual Memory Access on the VMEbus. All the DVMA requests from vme device drivers are completely handled by the driver. A DVMA request can be made for a maximum of 1624 Kilobytes at a time.

tvme(4)

DEVICES AND NETWORK INTERFACES

tvme(4)

FILES

/kernel drv tvme

SEE ALSO

vme 4

driver conf 4.

Themis Computer SPARC10MP Technical Manual

Themis Computer SPARC10MP Software Manual

NAME

vmedma - Themis driver for Direct Memory Access

SYNOPSIS

```
fd = open ( "/dev/vmedma0", O_RDWR );

err = ioctl (fd, command, arg);
```

DESCRIPTION

On Themis Computer's SPARC10MP platforms, the Fujitsu MB86366 MBus to VMEbus Interface Controller (MVIC) features a high performance DMA controller that provides bi-directional memory-to-VMEbus Direct Memory Access data transfers. The vmedma driver provides user programs access to the Direct Memory Access engine of the MVIC. The DMA engine can support all these transfers:

- memory to VMEbus
- VMEbus to memory
- memory to memory

The vmedma driver is designed to provide a powerful mechanism for highly efficient transfers involving the memory and the VMEbus. The user interface is uniform across the different hardware platforms available from Themis Computer. The vmedma driver is configured as a child of the VMEbus nexus driver and will function only on Themis Computer platforms.

The driver supports the open(2), close(2), ioctl(2) and poll(2) system calls. The commands implemented by the driver are accessed through the ioctl(2) system call. The VIOCDMACOPY ioctl command implements the basic DMA transfer capability of the driver. In addition to transfers between VMEbus and memory, the vmedma driver also supports transfers from memory to memory and VMEbus to VMEbus. Depending on the capabilities of the hardware platform, these transfers may be emulated in software. The VIOCSETCAP ioctl command may be used to interrogate the capabilities of the underlying hardware mechanism.

The driver supports the poll(2) system call for the event POLLRDNORM. The poll(2) system call blocks till the hardware DMA engine is available and returns when the engine has completed all queued transfers. However, a successful return of poll(2) does not guarantee that a subsequent VIOCDMACOPY command issued by the thread will not block.

IOCTLS

VIOCDMACOPY	This command performs a Direct Memory Access transfer between the specified address locations. The source address, destination address, size of the transfer and the type of
-------------	--

transfer are specified in a `vme_dmacopy` structure. A pointer to this structure is passed as the argument to the `ioctl(2)` call. The user program is suspended till the DMA transfer is complete, unless the `VF_NOBLOCK` flag is specified. The third argument of the `ioctl(2)` call is a pointer to a struct `vme_dmacopy`, which is defined in `<themis/vme_dmaio.h>` as

9

```

struct vme_dmacopy {
    u_int dma_flags; /* flags */
    void *dma_source; /* source address */
    void *dma_dest; /* destination address */
    void *dma_sourcehi; /* high bits in source address */
    void *dma_desthi; /* high bits in dest. address */
    u_int dma_count; /* byte count */
    struct vme_dmaresult *
        dma_result; /* pointer to result area */
};

struct vme_dmaresult {
    int dma_status; /* status of request */
    void *dma_bufid; /* retained buffer ID */
    int dma_errno; /* errno value */
}

```

9

`dma_source` and `dma_dest` specify the source and destination addresses, respectively, of the DMA transfer. If any or both of source and destination addresses are 64-bit VMEbus addresses, the `dma_sourcehi` and `dma_desthi` specify the corresponding higher bits of the addresses. The addresses are assumed to be user virtual addresses unless indicated otherwise by the `dma_flags` field.

The `dma_flags` member contains various flags which are OR'ed together to form the request. The first several flags are used to control the direction and type of the DMA transfer. The interpretation of the `dma_source` and `dma_dest` members is based on the value of `dma_flags`.

<code>VF_SVME</code>	The A32 VMEbus is the source of data for the transfer. If this flag is present, the <code>dma_source</code> value is an
----------------------	---

A32 VMEbus address

VF_SVME24	The A24 VMEbus is the source of data for the transfer. If this flag is present, the dma_source value is an A24 VMEbus address.
VF_SVME64	The A64 VMEbus is the source of data for the transfer. If this flag is present, the dma_source value is the low-order 32 bits of the A64 VMEbus address, and the dma_sourcehi value is the high-order 32 bits of the A64 VMEbus address.
VF_SVME16	The VMEbus source address refers to a 16 device.
VF_SVME64	The VMEbus source address refers to a 64 device.
VF_DVME	The VMEbus is the destination of data for the transfer. If this flag is present, the dma_dest value is an A32 VMEbus address.
VF_DVME24	The A24 VMEbus is the destination of data for the transfer. If this flag is present, the dma_dest value is an A24 VMEbus address.
VF_DVME64	The A64 VMEbus is the destination of data for the transfer. If this flag is present, the dma_dest value is the low-order 32 bits of the

A64 VMEbus address, and the `dma_desthi` value is the high-order 32 bits of the A64 VMEbus address

VF_DVMED16 The VMEbus destination address refers to a D16 device

VF_DVMED64 The VMEbus destination address refers to a D64 device

Additional flags are used to control the transfer as follows

VF_BLT Use block transfer mode

VF_NOWAIT If the DMA engine is busy, return an `EBUSY` error rather than queuing the request.

VF_NOBLOCK Return immediately after queuing or starting the request. The `vme_dmareresult` area will be updated when the transfer completes. Additionally, the user process can request to be notified via `SIGIO` that the transfer has completed. See below for more details

VF_RETAINBUFFER User memory buffer(s) are locked into memory. Any setup work that can be retained between uses of the buffer is retained. This may make repetitive transfers of small sizes more efficient. Specifying the flag results in valuable system resources being locked up. Hence this flag should be used

with care

VF_SIGDONE	signal when done (non-blocking requests only)
VF_SIGALL	signal all state changes (nonblocking requests only)

In all cases the address values must meet alignment restrictions. On SPARC10MP the addresses must be aligned on a 16-byte boundary. The dma_count member specifies the byte count of the transfer. It must be a multiple of sixteen bytes. The dma_result member points to a structure which is used to report information about the DMA transfer.

```
9 struct vme_dmares1 {
    int dma_status /* status of request */
    void *dma_bufid, /* retained buffer ID */
    int dma_errno, /* errno value */
};
```

9 The dma_status member contains a single value which represents the state of the transfer

VS_WAITING	request is waiting for the DMA engine
VS_INPROG	request is in progress
VS_DONE	request has finished normally
VS_ERROR	request has terminated due to error

If a request terminates due to an error, the dma_errno member will contain an appropriate errno value. If the VF_NCBLOCK flag is not used, this is the same as the errno returned from the ioctl(2) call.

If the VF_RETAINBUFFER flag is used, then dma_bufid will contain a buffer handle which

should be used in future VIOCDMACOPY calls and then released by issuing a VIOCRLBUF ioctl(2) command. When using VF_RETAINBUFFER, the dma_result structure pointer is required. The dma_bufid value of the structure should be initialized to 0 (null). In future requests using the same buffer or a subset thereof, the dma_bufid value should contain the value placed there by the first call. When the buffer is no longer needed, VIOCRLBUF must be called with the returned dma_bufid value to release the resources the retained buffer is using.

If the VF_NCBLOCK flag is used, then the result area is updated in real time. It is important the vme_dmaresult structure be allocated in static memory. Depending on the setting of the VF_SIGDONE and VF_SIGALL flags, SIGIO is sent to the process when the status changes (VF_SIGALL) or when the request is complete (VF_SIGDONE). When a request completes, SIGIO is sent to the process which made the request. The process must then inspect the result area(s) of its outstanding request(s) to determine that the request(s) did complete (with or without error). A SIGIO will be posted for every completed request (VF_SIGDONE) or status change (VF_SIGALL). However, signals are not queued, so the driver should inspect all outstanding result areas whenever a SIGIO is received.

VIOCRLBUF

This ioctl is used to release a buffer that was previously retained by specifying the VF_RETAINBUFFER flag for the VIOCDMACOPY command. The argument to the command is the dma_bufid in the vme_dmaresult structure returned by a previous VIOCDMACOPY command. If the value passed in is not a valid buffer handle, EINVAL is returned. If there is a problem releasing the buffer, EFAULT is returned. A problem releasing the buffer indicates an internal driver error and should be reported to Themis Technical Support.

VIOCGETCAP

This ioctl is used to determine the capabilities of the underlying hardware. Note that

THEMIS COMPUTER

Last change 19 Mar 1996

6

in most cases, if the hardware is incapable of performing a request via DMA then the work represented by the request will be performed via other mechanisms. This ensures a common interface for user programs across different hardware platforms available from Themis Computer. If the user programs wish to tailor the request to the hardware platform, they may do so by using the `VIOCAP` command.

The third argument to the `ioctl()` call is a pointer to a `vme_dmacap` structure which is filled in upon return from the call.

```

struct vme_dmacap {
    u_int dmac_flags;    /* flags */
    u_int dmac_align;    /* alignment modulus */
    u_int dmac_min;      /* minimum effective size */
};

```

The `dmac_flags` member will contain zero or more of the following flags, OR'ed together:

<code>VC_MEXTOMEM</code>	Memory-to-memory transfers are done with a DMA engine. Otherwise they are emulated with polled I/O.
<code>VC_VMETOVME</code>	VMEbus-to-VMEbus transfers are done with a DMA engine. Otherwise they are emulated with polled I/O.
<code>VC_VMETOMEM</code>	VMEbus-to-memory transfers are done with a DMA engine. Otherwise, they are emulated with polled I/O.
<code>VC_MEXTOVME</code>	Memory-to-VMEbus transfers are done with a DMA engine. Otherwise they are emulated with polled I/O.

<code>VC_D16</code>	D16 transfers are DMA transfers. If this flag
---------------------	---

is not present and a D16 transfer is requested, it will be done via polled I/O

VC_A64 VMEbus A64 accesses are supported. If this flag is not present and an A64 request is made, EINVAL will be returned

VC_D64 VMEbus D64 accesses are supported. If this flag is not present and a D64 request is made, EINVAL will be returned

The `dmac_align` member contains the alignment modulus that must be used for the address and count when making requests. A misaligned request will provoke an EINVAL error return

The `dmac_min` member contains a hint as to the minimum size of a DMA transfer. Transfers below the minimum size may be better accomplished via other system calls like `read(2)`, `write(2)` etc

Example code :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <string.h>
#include <sys/errno.h>
#include <stdlib.h>

#include <themis/vmedmaio.h>

/*
```

THEMIS COMPUTER Last change: 19 Mar 1996

6

vmedma(4) DEVICES AND NETWORK INTERFACES vmedma(4)

- * This is a sample program to illustrate the use of the vmedma driver
- * The program transfers 1000 words to VME address 0x72a00000


```

*,
    dma dma_source = to_buf, /* DMA transfer is from memory to VMEbus
    dma dma_dest = (void *)0x72a00000
    dma dma_sourcehi = dma dma_desthi = 0 /* no 64-bit addresses */
    dma dma_count = 1024 * sizeof(long) /* length of data transfer */
/
/
    dma dma_result = NULL, /* we don't bother about the result area */

    dma.dma_flags = VF_DVME, /* destination is VME; no other special
stuff */

    if ( ioctl(fd, VIODMACOPY, &dma) < 0) {
        (void) fprintf(stderr,
            "%s can't do DMA transfer %s0,
            argv[0], sys_errlist[errno]);
        free ( to_buf);
        exit 1
    }

    free ( to_buf);

    /* allocate destination memory buffer */
    if ((from_buf = (long int *)malloc ( 1024 * sizeof(long))) == NULL
    {
        (void) fprintf(stderr,
            "%s: can't allocate destination memory buffer: %s0,
            argv[0], sys_errlist[errno]);
        exit 1
    }

    /* initialise the memory buffer with numbers from 0 to 1023 */
    for ( i = 0; i < 1024; i++)
        from_buf[i] = i;

    /* set up the DMA request
    *- VMEbus to memory
    *- 32-bit data, 32-bit address
    *- no B-T
    *- do not retain buffers
    */
    dma dma_source = (void *) 0x72a00000,
    dma dma_dest = (void *) from_buf; /* DMA transfer is from VMEbus
to memory */

    dma dma_sourcehi = dma dma_desthi = 0 /* no 64-bit addresses */
    dma.dma_count = 1024 * sizeof(long) /* length of data transfer */
/
/
    dma dma_result = NULL; /* we don't bother about the result area */

    dma.dma_flags = VF_SVME; /* source is VME; no other special stuff

    if (ioctl(fd, VIODMACOPY, &dma) < 0)
    {

```

```

(void) fprintf(stderr,
"%s can't do DMA transfer %s0,
    argv[0], sys_errlist(errno)),
free ( from_buf),
exit 1'
}

/* now compare what we read with what we wrote */
for ( i = 0; i < 1024; i++)
if ( from_buf[i] != i )
printf
"Data check error at offset %x: got %x, expected %x0,
i, from_buf[i], i)

/* clean up and return */
free ( from_buf),
close ( fd),

}

```

FILES

```

dev vmedma0
/usr/include/thesis/vmectl.h
/kernel/drv/vmedma.conf

```

SEE ALSO

```

tyme(7,
vme 4,
driver.conf(4)
Thesis Computer SPARC10MP User Manual
Thesis Computer SPARC10MP Software Manual

```

NAME

vmcdvma - Themis sample driver for Direct Virtual Memory Access

SYNOPSIS

```
#include <themis/vmcdvma.h>

fd = open ( "/dev/vmcdvma0", O_RDWR),

err = ioctl( fd, cmd, arg);
```

DESCRIPTION

vmcdvma is a sample driver provided by Themis Computer to illustrate the use of Direct Memory Access within a device driver. The driver is provided along with the source code to help developers writing vme device drivers. The driver will function only on Themis Computer platforms.

The user program invokes the driver by opening the device and issuing the appropriate ioctl() calls as needed. In addition to the open() and ioctl() system calls, the driver supports the close() system call to close the driver instance. Through the ioctl() commands provided, the user can allocate a number of DVMA regions on the VMEbus. The user can write to or read from these regions and free them.

IOCTL INTERFACE

The third argument to ioctl() is a pointer to a struct vme_dvmacopy, which is defined in <themis/vmcdvma.h> as

```
struct vme_dvmacopy {
    int      size;
    int      id;
    caddr_t  addr;
    int      offset;
};
```

The size and offset parameters is specified by the application and is not modified by the driver. The addr parameter is used as a read/write parameter and can be changed by the application. The id member is used to identify the DVMA region and should not be modified by the application.

VOMA_ALLOCATE allocates a DMA region. The input parameter size specifies the size of the region to be allocated. The driver allocates a DVMA region and sets the id parameter. The address of the DVMA region is returned in the addr parameter. If a DMA region can not be allocated because of memory constraints, the ioctl() call fails and sets errno to ENOMEM. The id parameter is used to identify the allocated

VDMA region For all further operations on the region. Any other board on the VME chassis can access the allocated VDMA region using the `addr` parameter

VDMA_FREE frees the VDMA region specified by the `id` parameter. All resources allocated for the region will be freed

VDMA_WRITE copies data from user memory to the VDMA region. The `id` parameter must specify a value previously returned by a **VDMA_ALLOCATE**. The `addr` parameter specifies the user memory location from where the data will be copied. The `size` parameter specifies the amount of data to be copied. The `offset` parameter specifies the location from the beginning of the VDMA region where user data will be written to. The sum of `size` and `offset` must not be greater than the size of the allocated region

VDMA_READ copies data from the VDMA region to user memory. The `id` parameter must specify a value previously returned by a **VDMA_ALLOCATE**. The `addr` parameter specifies the user memory location into which data will be copied. The `size` parameter specifies the amount of data to be copied. The `offset` parameter specifies the location from the beginning of the VDMA region from where the data will be read. The sum of `size` and `offset` must not be greater than the size of the allocated region

Example code .

This program allocates a VDMA region that is 26 bytes long. It then writes the lower case alphabets into the region one character at a time. It then reads the data and compares them with what was written.

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

#include <themis/vmedvma.h>

main
{
    int fd;
    int count;
    struct vme_dvmacopy dma_req;
    struct vme_dvmacopy dma_read;
    struct vme_dvmacopy dma_write;
    char alpha[26];
```

```

char beta[26].

/* open the device */
if ((fd = open("/dev/vmedvma0", O_RDWR)) < 0)
{
    perror("can't open /dev/vmedvma0");
    exit(1);
}

/* allocate a DVMA region */
dma_req size = 26;
if (ioctl(fd, VDMA_ALLOCATE, &dma_req) != 0)
{
    perror("In allocating DVMA");
    close(fd);
    return(-1);
}

printf("Allocated a DVMA region id : %d, at virtual sbus address : %x0, dma_req id, dma_req addr)

/* initialise the dma write request */
dma_write id = dma_req id
dma_write size = 1;

/* write the alphabet character by character */
for (count = 0, count < 26, count++)
{
    alpha[count] = 'a' + count;
    dma_write addr = alpha-count;
    dma_write offset = count;

    if (ioctl(fd, VDMA_WRITE, &dma_write) != 0)
    {
        perror("In writing");
        close(fd);
        return -1;
    }
}

/* initialise the dma read request */
dma_read id = dma_req id
dma_read size = 26;
dma_read addr = beta
dma_read offset = 0

if (ioctl(fd, VDMA_READ, &dma_read) != 0)
{
    perror("In reading");
    close fd;
    return -1;
}

```



```
/* read the DVMA memory and compare with what was written */
for ( count = 0, count < 26; count ++ )
{
    if ( alpha[count] != beta[count] )
        printf("Data mismatch: Wrote %c, Read %c0,
alpha[count], beta[count])
}

/* clean up and return */
close(fd);
return 0);
}
```

FILES

```
.dev vmedvma0
usr/include/thesis/vmedvma.h
.kernel driver vmedvma.conf
```

SEE ALSO

```
tvme 7)
vme 4
driver.conf 4
Thesis Computer SPARC10MP Technical Manual
Thesis Computer SPARC10MP Software Manual
```

NAME

vmeintr - Themis sample driver for VMEbus interrupts

SYNOPSIS

```
#include <themis/vmeintr.h>

fd = open ( /dev vmeintr0 ,

err = ioctl( fd, SETINTR, arg);
```

DESCRIPTION

vmeintr is a sample driver provided by Themis Computer to illustrate the vectored interrupt mechanism of the VMEbus. The driver is provided along with the source code to help developers writing vme device drivers. The driver will function only on Themis Computer platforms.

The driver provides a mechanism by which an interrupt can be generated on the VMEbus. This interrupt can be received by any other VME board on the VMEbus. The system that generates the interrupt would not receive it. The vmeintr driver relies on the driver configuration files to specify the interrupts it should handle. Each instance of the driver registers the interrupts with the system. Through the ioctls implemented by the driver, the user can generate interrupts and send them to the appropriate driver instances.

The user program invokes the driver by opening the device and issuing the appropriate ioctl() calls as needed. In addition to the open() and ioctl() system calls the driver supports the close() system call to close the driver instance.

IOCTL INTERFACE

The third argument to ioctl() is a pointer to a struct vme_intr_vect, which is defined in <themis/vmeintr.h> as

```
typedef struct {
    int vector;
    int priority;
    long timeout; /* used only for GETINTRDATA */
} vme_intr_vect;
```

vector specifies the VMEbus interrupt vector and has a value between 0 - 255. priority specifies the VMEbus interrupt priority and has a value between 1 - 7. timeout is used only for the GETINTRDATA command and specifies the time in microseconds.

REQINTRSIGNAL prepares the driver to receive the specified interrupt from another board on the VME chassis. The

priority - vector pair should be a valid value specified in the configuration file. An invalid value will result in an error of EINVAL being returned.

SENDINTR sends the specified interrupt on the VMEbus. This interrupt will be received by any board on the VMEbus that has the particular interrupt enabled.

GETINTRDATA waits for the specified interrupt. If the interrupt has already been received, the call will return immediately. Otherwise the calling process will be blocked till the interrupt is received or till timeout microseconds elapse, whichever occurs earlier. If the timeout is specified as -1, the calling process will be blocked till the interrupt is received.

Typically two systems sharing the same VME chassis are needed to use the vmaintr driver. One of the systems is used to generate the interrupts and another is used to receive the interrupts.

Example code (interrupt receiver)

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

#include <themis/vmaintr.h>

int fd,
vme_intr_vect vect,
vme_intr_vect vect_recvd,

if (fd = open("/dev/vmaintr0", O_RDWR)<0)
{
    perror("can't open /dev/vmaintr0");
    exit(1);
}

/* assume argv[1] specifies the vector and argv[2] specifies
the priority
*/
vect.vector = atoi ( argv[1] );
vect.priority = atoi ( argv[2] ),
```

```

/* enable the interrupt */
if (ioctl(fd, REQINTRSIGNAL, &vect) < 0)
{
    perror("In enabling interrupts"),
    close(fd),
    return(-1),
}

/* wait for the interrupt */
printf("Waiting for interrupt : %d , 0, vect vector,,\n", vect);
vect_rcvd timeout = 30 * 1000 * 1000; /* wait for 30 seconds */
if (ioctl(fd, GETINTRDATA, &vect_rcvd) < 0)
{
    if (errno == ETIME)
        printf("Timeout occurred. Interrupt was probably not generated\n");
    else
        perror("In receiving interrupts");
    close(fd),
    return(-1),
}

/* print the received interrupt */
printf("Received vector %d at priority0, vect_rcvd.vector,\n", vect_rcvd.priority);

/* clean up and return */
close(fd);
return(0);

```

Example code { interrupt generator:

```

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

#include <themis/vmeintr.h>

int fd,
vme_intr_vect vect,

if ((fd = open("/dev/vmeintr0", O_WRONLY)) < 0)
{

```

```

    perror("can't open /dev/vmeintr0"),
    exit 1
}

/* assume argv[1] specifies the vector and argv[2] specifies
   the priority
*/

vect vector = atoi ( argv[1] );
vect priority = atoi ( argv[2] );

/* send the interrupt */
printf("Sending interrupt : %d .. 0, vect vector,
if (ioctl(fd, SENDINTR, &vect) < 0)
{
    perror("In sending interrupts");
    close(fd);
    return(-1);
}

/* clean up and return */
close fd;
return(0);

```

```

/dev/vmeintr0
/usr/include/themis/vmeintr.h
/kernel.drv/vmeintr.conf

```

SEE ALSO

evme 7)

vme 4

driver.conf 4

Themis Computer SPARC10MP User Manual

Themis Computer SPARC10MP Software Manual

NAME

vmemem - Device driver for VMEbus access

SYNOPSIS

```
fd = open ( "/dev/vmeXXXXY", O_RDWR ),
```

DESCRIPTION

The vmemem driver provides a standard way to access the VMEbus from user processes. The driver creates device special files under the /dev directory. These files support open(), close(), read(), write() and mmap() calls to access the VMEbus. The type of access to the VMEbus is indicated by the name of the device file.

```
/dev/vme12d32 - 32 bit address, 32 bit data /dev/vme12d16
- 32 bit address, 16 bit data /dev/vme24d32 - 24 bit
address, 32 bit data /dev/vme24d16 - 24 bit address, 16
bit data /dev/vme16d32 - 16 bit address, 32 bit data
/dev/vme16d16 - 16 bit address, 16 bit data
```

The user program accesses any part of the VMEbus address space by opening the device file and accessing it at a particular offset. For example, to access address location 0x72000000 of the 32 bit VME space, the user can open the file /dev/vme12d32 and seek to offset 0x72000000. For efficient access, the user can map the VMEbus address space into the user address space by using the mmap() system call.

This program accesses a specific region in the A32 address space. It initialises the region and reads the values back.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main
{
    int fd;
    int count;
    int value;

    /* open the device */
    if ((fd = open("/dev/vme12d32", O_RDWR)) < 0)
    {
```

```

        perror("can't open /dev/vmemem");
        exit 1
    }

    lseek( fd, 0x72000000, SEEK_SET);

    value = 0
    for ( count = 0; count < 1024; count ++ )
        if ( write ( fd, &value, sizeof(int)) != sizeof(int) )
            perror "In writing ";

    lseek( fd, 0x72000000, SEEK_SET);

    for ( count = 0; count < 1024; count ++ )

        if ( read ( fd, &value, sizeof(int)) != sizeof(int) )
            perror("In reading");
        else
            if ( value != 0 )
                printf("Error at. %x, read %d instead of zero0,
                    0x72000000 + (count * sizeof(int)), value),
        }

    close fd
    return(0);
}

```

FILES

```

/dev/vme12d32
/dev/vme12d16
/dev/vme14d32
/dev/vme14d16
/dev/vme16d32
/dev/vme16d16
/kernel drv/vmemem.conf

```

SEE ALSO

```

byme +

```

vmemem(4)

DEVICES AND NETWORK INTERFACES

vmemem(4)

vme 4

driver conf 4

Themis Computer SPARC10MP Technical Manual

Themis Computer SPARC10MP Software Manual

simplifiedma 4) DEVICES AND NETWORK INTERFACES simplifiedma(4)

NAME

simplifiedma - Simplified sample driver for Direct Memory Access

SYNOPSIS

```
fd = open ( "/dev/simplifiedma0", O_RDWR ),
err = ioctl (fd, command, arg
```

DESCRIPTION

simplifiedma is a sample driver provided by Themis Computer to illustrate the use of Direct Memory Access from drivers for VMEbus devices. The driver is a simplified version of the more powerful vmedma driver. The driver is provided along with the source code. On Themis Computer's SPARCCLX+ and SPARC5/64 platforms, the Newbridge SC764 VMEbus Controller (SCV64) features a high performance DMA controller that provides bi-directional memory-to-VMEbus Direct Memory Access data transfers. The VMEbus nexus driver exports a set of functions that can be used by VME leaf drivers to do Direct Memory Access operations on the VMEbus. The simplifiedma driver illustrates the use of these functions. The simplifiedma driver is configured as a child of the VMEbus nexus driver and will function only on Themis Computer platforms.

The driver supports the open(2), close(2), ioctl(2) and poll(2) system calls. The commands implemented by the driver are accessed through the ioctl(2) system call. The VIOCDMACOPY ioctl command implements the basic DMA transfer capability of the driver. In addition to transfers between VMEbus and memory, the simplifiedma driver also supports transfers from memory to memory and VMEbus to VMEbus. Depending on the capabilities of the hardware platform, these transfers may be emulated in software. The VIOCGETCAP ioctl command may be used to interrogate the capabilities of the underlying hardware mechanism.

The driver supports the poll(2) system call for the event POLLRDNORM. The poll(2) system call blocks till the hardware DMA engine is available and returns when the engine has completed all queued transfers. However a successful return of poll(2) does not guarantee that a subsequent VIOCDMACOPY command issued by the thread will not block.

IOCTLS

VIOCDMACOPY	This command performs a Direct Memory Access transfer between the specified address locations. The source address, destination address, size of the transfer and the type of
-------------	--

transfer are specified in a `vme_dmacopy` structure. A pointer to this structure is passed as the argument to the `ioctl(2)` call. The user program is suspended till the DMA transfer is complete, unless the `VF_NOBLOCK` flag is specified. The third argument of the `ioctl(2)` call is a pointer to a structure `vme_dmacopy`, which is defined in `<themis/vmedmaio.h>` as

9

```

struct vme_dmacopy {
    u_int dma_flags,      /* flags */
    void *dma_source,     /* source address */
    void *dma_dest,       /* destination address */
    void *dma_sourcehi,   /* high bits in source address */
    void *dma_desthi,     /* high bits in dest. address */
    u_int dma_count,      /* byte count */
    struct vme_dmareult *
    dma_result,           /* pointer to result area */
};

struct vme_dmareult {
    int dmar_status,      /* status of request */
    void *dmar_buf_id,    /* retained buffer ID */
    int dmar_errno,       /* errno value */
};

```

9

`dma_source` and `dma_dest` specify the source and destination addresses, respectively, of the DMA transfer. If any or both of source and destination addresses are 64-bit VMEbus addresses the `dma_sourcehi` and `dma_desthi` specify the corresponding higher bits of the addresses. The addresses are assumed to be user virtual addresses unless indicated otherwise by the `dma_flags` field.

The `dma_flags` member contains various flags, which are OR'ed together to form the request. The first several flags are used to control the direction and type of the DMA transfer. The interpretation of the `dma_source` and `dma_dest` members is based on the value of `dma_flags`.

VF_SVME

The A32 VMEbus is the source of data for the transfer. If this flag is present, the `dma_source` value is an

A32 VMEbus address

VF_SVME24	The A24 VMEbus is the source of data for the transfer. If this flag is present, the dma_source value is an A24 VMEbus address.
VF_SVME64	The A64 VMEbus is the source of data for the transfer. If this flag is present, the dma_source value is the low-order 32 bits of the A64 VMEbus address and the dma_sourcehi value is the high-order 32 bits of the A64 VMEbus address.
VF_SVME0.6	The VMEbus source address refers to a D.6 device.
VF_SVME0.64	The VMEbus source address refers to a D.64 device.
VF_DVME	The VMEbus is the destination of data for the transfer. If this flag is present, the dma_dest value is an A32 VMEbus address.
VF_DVME24	The A24 VMEbus is the destination of data for the transfer. If this flag is present, the dma_dest value is an A24 VMEbus address.
VF_DVME64	The A64 VMEbus is the destination of data for the transfer. If this flag is present, the dma_dest value is the low-order 32 bits of the

A64 VMEbus address, and the `dma_desthi` value is the high-order 32 bits of the A64 VMEbus address

VF_DMED16 The VMEbus destination address refers to a D16 device

VF_DMED64 The VMEbus destination address refers to a D64 device

Additional flags are used to control the transfer as follows

VF_BLT Use block transfer mode

VF_NOWAIT If the DMA engine is busy, return an `EBUSY` error rather than queuing the request

VF_NOBLOCK Return immediately after queuing or starting the request. The `vme_drareault` area will be updated when the transfer completes. Additionally the user process can request to be notified via `SIGIO` that the transfer has completed. See below for more details

VF_SIGDONE signal when done (non-blocking requests only)

VF_SIGALL signal all state changes (nonblocking requests only)

In all cases, the address values must meet alignment restrictions. On `SPARC3/64` the addresses must be aligned on a 8-byte boundary. The `dma_count` member specifies the

byte count of the transfer. It must be a multiple of eight bytes. The dma_result member points to a structure which is used to report information about the DMA transfer.

```

struct vme_dmaresult {
    int dma_status;      /* status of request */
    void *dma_bufid;     /* retained buffer ID */
    int dma_errno;       /* errno value */
}

```

The dma_status member contains a single value which represents the state of the transfer.

VS_WAITING	request is waiting for the DMA engine
VS_INPROG	request is in progress
VS_DONE	request has finished normally
VS_ERROR	request has terminated due to error

If a request terminates due to an error, the dma_errno member will contain an appropriate errno value. If the VF_NCBLOCK flag is not used, this is the same as the errno returned from the ioctl(2) call.

If the VF_NCBLOCK flag is used, then the result area is updated in real time. It is important the vme_dmaresult structure be allocated in static memory. Depending on the setting of the VF_SIGDONE and VF_SIGALL flags, SIGIO is sent to the process when the status changes (VF_SIGALL) or when the request is complete (VF_SIGDONE). When a request completes, SIGIO is sent to the process which made the request. The process must then inspect the result area(s) of its outstanding request(s) to determine that the request(s) did complete (with or without error). A SIGIO will be posted for every completed request (VF_SIGDONE, or status change (VF_SIGALL). However, signals are not queued, so the driver should inspect all outstanding result areas whenever a SIGIO is

received

VIOCGSTCAP

This ioctl is used to determine the capabilities of the underlying hardware. Note that in most cases, if the hardware is incapable of performing a request via DMA (such as memory-to-memory DMA transfer on SPARC5/64) then the work represented by the request will be performed via other mechanisms. This ensures a common interface for user programs across different hardware platforms available from Themis Computer. If the user programs wish to tailor the request to the hardware platform, they may do so by using the VIOCGTCA? command.

The third argument to the ioctl() call is a pointer to a vme_dmacap structure which is filled in upon return from the call.

```
struct vme_dmacap {
    u_int dmac_flags;    /* flags */
    u_int dmac_align;    /* alignment modulus */
    u_int dmac_min;      /* minimum effective size */
}
```

The dmac_flags member will contain zero or more of the following flags, OR'ed together.

VC_MEMTOMEM	Memory-to-memory transfers are done with a DMA engine. Otherwise they are emulated with polled I/O.
VC_VMETOVME	VMEbus-to-VMEbus transfers are done with a DMA engine. Otherwise, they are emulated with polled I/O.
VC_VNETOMEM	VMEbus-to-memory transfers are done with a DMA engine. Otherwise they are emulated with polled I/O.

THEMIS COMPUTER

Last change 19 Mar 1996

6

simpledma(4)

DEVICES AND NETWORK INTERFACES

simpledma(4)

VC_MEMTOVME Memory-to-VMEbus

transfers are done with a DMA engine. Otherwise, they are emulated with polled I/O

VC_D16 D16 transfers are DMA transfers. If this flag is not present and a D16 transfer is requested, it will be done via polled I/O

VC_A64 VMEbus A64 accesses are supported. If this flag is not present and an A64 request is made, EINVAL will be returned

VC_D64 VMEbus D64 accesses are supported. If this flag is not present and a D64 request is made, EINVAL will be returned

The `dma_align` member contains the alignment modulus that must be used for the address and count when making requests. A misaligned request will provoke an EINVAL error `re-r7`

The `dma_min` member contains a hint as to the minimum size of a DMA transfer. Transfers below the minimum size may be better accomplished via other system calls like `read(2)`, `write(2)` etc

Example code :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
```

```
#include <string.h>
#include <sys/errno.h>
```

```

#include <stdlib.h>

#include <themis/vme_dma10.h>

/*
 * This is a sample program to illustrate the use of the simplifiedma driver.
 *
 * The program transfers 1024 words to VME address 0x72a00000
 * The program then reads the VME memory and verifies that the
 * data was copied correctly
 */
/*
 * External data
 */
extern char *sys_errlist[]; /* error strings */
extern int errno;

main(int argc, char * argv[])
{
    struct vme_dmacopy dma; /* DMA request structure */
    int i; /* loop counter */
    struct vme_dmarresult *dmar = 0; /* result structure */
    int fd; /* VME DMA engine fd */
    struct timeval start, end; /* timing */
    int ns; /* # of reqs started */
    int err = 0; /* error flag */
    double kb; /* kilobytes copied */
    double secs; /* # of seconds */
    long int * to_buf, * from_buf; /* memory buffers */

    /* open the DMA engine */
    if ((fd = open("/dev/simplifiedma0", O_RDWR)) < 0)
    {
        (void) fprintf(stderr,
        "%s can't open /dev/simplifiedma0 %s0,
        argv[0] sys_errlist(errno),
        exit(1),
        }

        /* allocate source memory buffer */
        if ((to_buf = (long int *)malloc (1024 * sizeof(long))) == NULL)
        {
            (void) fprintf(stderr,
            "%s can't allocate source memory buffer: %s0,
            argv[0], sys_errlist(errno),
            exit(1),
            }

```

THEMIS COMPUTER

Last change 19 Mar 1996

8

simplifiedma(4)

DEVICES AND NETWORK INTERFACES

simplifiedma(4)

```

/* initialise the memory buffer with numbers from 0 to 1023 */
for (i = 0, i < 1024, i++)

```



```

to_buf[i] = i;

/* set up the DMA request
 * - memory to VMEbus
 * - 32-bit data, 32-bit address
 * - no BLT
 * - do not retain buffers
 */

dma.dma_source = to_buf; /* DMA transfer is from memory to VMEbus
*/
dma.dma_dest = (void *)0x72a00000;
dma.dma_sourcehi = dma.dma_desthi = 0; /* no 64-bit addresses */
dma.dma_count = 1024 * sizeof(long); /* length of data transfer */
/
dma.dma_result = NULL; /* we don't bother about the result area */
/

stuff */

dma.dma_flags = VF_DVME; /* destination is VME; no other special

if ( ioctl(fd, VIODMACOPY, &dma) < 0) {
(void) fprintf(stderr,
"%s: can't do DMA transfer: %s0,
    argv[0], sys_errlist[errno]);
free ( to_buf);
exit(1);
}

free ( to_buf);

/* allocate destination memory buffer */
if ((from_buf = (long int *)malloc ( 1024 * sizeof(long))) == NULL)
{
(void) fprintf(stderr,
"%s: can't allocate destination memory buffer: %s0,
    argv[0], sys_errlist[errno]);
exit(1);
}

/* initialize the memory buffer with numbers from 0 to 1023 */
for ( i = 0; i < 1024; i++)
from_buf[i] = i;

/* set up the DMA request
 * - VMEbus to memory
 * - 32-bit data, 32-bit address
 * - no BLT
 * - do not retain buffers
 */
dma.dma_source = (void *) 0x72a00000;
dma.dma_dest = (void *) from_buf; /* DMA transfer is from VMEbus
to memory */
dma.dma_sourcehi = dma.dma_desthi = 0; /* no 64-bit addresses */

```

```

        dma.dma_count = 1024 * sizeof(long); /* length of data transfer */
        dma.dma_result = NULL; /* we don't bother about the result area */

        dma.dma_flags = VF_SVME; /* source is VME; no other special stuff */

        if (ioctl(fd, VIODMACOPY, &dma) < 0)
        {
            (void) fprintf(stderr,
                "%s: can't do DMA transfer: %s0,
                argv[0], sys_errlist(errno));
            free (from_buf);
            exit(1);
        }

        /* now compare what we read with what we wrote */
        for ( i = 0; i < 1024; i++)
            if ( from_buf[i] != i )
                printf(
                    "Data check error at offset %x: got %x, expected %x0,
                    i, from_buf[i], i);

        /* clean up and return */
        free ( from_buf);
        close ( fd);
    }
}

```

FILES

```

    /dev/simplifiedma0
    /usr/include/themis/vmectlio.h
    /kernel/drv/simplifiedma.conf

```

SEE ALSO

```

    tvme(7)
    vme(4)
    driver.conf(4)
    Themis Computer SPARC10MP Technical Manual

```

Themis Computer SPARC10MP Software Manual

